

# FONDAMENTI DI PROGRAMMAZIONE IN PYTHON

FABIO PELLACINI

```
0101010001110101001001101001000100  
01001programmazione001001001110101  
01001tipi01001000101oggetti0010000  
001001variabili0010010funzioni0001  
0010010101sequenze00101dizionari00  
0010101immagini00101000testo110100  
0010000grafi00100100alberi00101001  
00101ricorsione111010ordinamento00  
001000111linguaggi001001python1110  
010010web00101cli0010gui0010010000  
00101interazione0010elaborazione11
```

# FONDAMENTI DI PROGRAMMAZIONE IN PYTHON

FABIO PELLACINI

VERSIONE 1.0.0

Copyright © 2016 Fabio Pellacini

Tutti i diritti riservati. Nessuna parte di questo libro può essere riprodotta, memorizzata o trasmessa senza l'esplicito consenso dell'autore. Il codice è rilasciato come open source sul sito [github.com/xelatihy/fondamentibook](https://github.com/xelatihy/fondamentibook).

Il contenuto di questo libro è distribuito senza garanzie di alcun tipo, esplicite o implicite, come ad esempio le garanzie di commerciabilità, idoneità ad un fine particolare o di non violazione dei diritti di altri. In nessun caso l'autore potrà essere ritenuto responsabile per qualsiasi reclamo, danno, o altro tipo di responsabilità, derivante o in connessione con il contenuto di questo libro, il suo utilizzo o altre attività relative allo stesso.

Nomi e marchi citati nel testo sono generalmente depositati o registrati dalle rispettive aziende. Alcune immagini contenute in questo libro sono in Public Domain e reperite sul sito di Wikipedia. Questo ebook è riprodotto con l'uso delle fonts open source Open Sans e Hack.

# PREFAZIONE

Questo libro introduce i principianti alle principali idee della programmazione utilizzando il linguaggio Python. Il mio scopo non è quello di presentare un trattamento completo di tutti i principi di programmazione né di coprire tutti gli aspetti del linguaggio stesso. Voglio invece mostrare, in modo succinto, come si possono affrontare problemi di programmazione reali.

Dopo una breve introduzione ai basilari concetti di programmazione, ogni capitolo considera un problema, ne discute come modellarlo al calcolatore, e introduce nuovi concetti di programmazione utili alla sua risoluzione. Ho scelto problemi da campi applicativi diversi per mostrare diversi stili di programmazione. Le soluzioni presentate sono succinte ma funzionali, implementate facendo un uso esteso della libreria standard di Python e di librerie esterne.

Questo libro è stato scritto in *literate programming*, uno stile di scrittura in cui il testo è un commento al codice di ogni capitolo. *Ogni capitolo è infatti un programma Python eseguibile* che, quando appropriato, calcola valori su dati scaricati da Internet. Questa è forse la differenza maggiore tra questo testo e la moltitudine di libri già presenti. Il codice per ogni capitolo è rilasciato sul sito [github.com/xelatihy/fondamentibook](https://github.com/xelatihy/fondamentibook).

Il materiale da cui questo libro deriva è stato usato come supporto all'insegnamento del corso di *Fondamenti di Programmazione* offerto dal Dipartimento di Informatica dell'Università di Roma La Sapienza. Ringrazio Riccardo Silvestri per l'aiuto nel redigere il materiale originale, e i colleghi Chierichetti, De Marsico, Melatti, e Wollan per avere supportato il corso.

Dopo avere testato questo metodo di insegnamento per vari anni in classe, ho deciso di pubblicarlo in forma di libro nella speranza che aiuti una nuova generazione di principianti nell'apprendimento della programmazione.

*Happy hacking!* Buona programmazione!

Settembre 2016

Fabio Pellacini

A handwritten signature in black ink, reading "Fabio Pellacini". The signature is written in a cursive style with a long horizontal line extending to the right.

*a Camilla*

# INDICE

## 1. I Computer e la Programmazione

L'Architettura degli Elaboratori. La Memoria e I Dati. La CPU e i Programmi. Linguaggi a Basso Livello. Linguaggi ad Alto Livello. Storia dei Linguaggi di Programmazione. Sviluppo Software. Programmazione Moderna.

## 2. Primi Passi in Python

Installazione di Python. Esecuzione di Programmi in Python. Programmazione Interattiva. Errori di Programmazione. Risorse per Programmare in Python.

## 3. Python come Calcolatrice

Espressioni Aritmetiche. Commenti. Stampa di Valori. Stringhe. Tipi e Conversioni. Variabili e Assegnamenti. Incrementi.

## 4. Riutilizzo di Istruzioni

Funzioni. Funzioni predefinite. Funzioni che Ritornano Più Valori. Parametri Opzionali. Moduli e File. Documentazione Interattiva.

## 5. Prendere Decisioni

Valori Booleani. Operatori Relazionali. Operatore di appartenenza. Istruzioni Condizionali. Moduli e Programmi.

## 6. Sequenze di Dati

Liste. Tuple. Operazioni su Sequenze. Operatori Relazionali e di Appartenenza. Elementi di Sequenze. *Unpacking* di Valori. Sottosequenze o *Slices*.

## 7. Iterazione su Sequenze

Iterazione su Sequenze. Iterazione su Sequenze di Interi. Iterazione con Condizionali. Controllo dell'Iterazione. Iterazione su Condizione. Iterazione e *Unpacking*. Comprehensions.

## 8. Oggetti e Metodi

Oggetti, Tipi e Identità. Oggetti Funzione. Assegnamento di Oggetti. Metodi. Metodi delle Liste.

## 9. Tabelle di Dati

Dizionari. Insiemi. Tabelle di Dati. Estrazione di Dati. Ricerca di Dati. Ordinamento di Dati.

## 10. Accesso ai Dati

File e Percorsi. Apertura di File. Codifica dei File di Testo. Lettura di File. Scrittura di File. Input e Output di Dati. Accedere a Documenti sul Web.

## 11. Elaborazione di Testo

Metodi delle Stringhe. Elaborazione del Testo. Ricerca di Documenti.

## 12. Elaborazione di Immagini

Rappresentazione dei Colori. Rappresentazione di Immagini. Salvataggio di Immagini. Creazione di Immagini. Accesso ai Pixel. Operazioni sui Colori. Caricamento di Immagini. Copie e Cornici. Rotazioni. Modifica dei Colori. Mosaici. Spostamento di Pixels.

## 13. Tipi Definiti dall'Utente

Classi. Costruttore. Oggetti. Metodi. Metodi speciali. Incapsulamento.

#### 14. Esplorare il File System

Esplorare il File System. Ricorsione. Alberi. Alberi di Oggetti.

#### 15. Documenti Strutturati

HTML. Rappresentazione di Documenti HTML. Parsing di Documenti HTML. Operazioni sui Documenti.

#### 16. Interfacce Utente

Programmi Interattivi. Librerie per Interfacce Utente. Applicazioni Qt. Metodi Statici. Widgets, Eventi e Callbacks. Layouts. Esempio: TextEditor. Esempio: WebBrowser.

#### 17. Grafica Interattiva

Scheletro dell'Applicazione. Disegno di Forme. Pulizia dell'Immagine. Interazione. Variabili Globali. Trasformazioni.

#### 18. Giochi

Stato del Gioco. Grafica. Aggiornamento del Gioco. Interazione.

#### 19. Simulazione Interattiva

Simulazione di Particelle. Vettori. Modello per le Particelle. Struttura del Programma. Simulazione del Moto. Parametri di Simulazione. Creazione di Particelle. Forze. Interazione. Generazione Continua. Collisioni.

#### 20. Applicazioni Web e CLI

Struttura dell'Applicazione. Funzionalità Logiche dell'Applicazione. Decoratori. Interfaccia a Riga di Comando. Interfaccia Web.

#### 21. Navigare Labirinti

Grafi. Rappresentazione di Grafi. Visualizzazione di Grafi. Labirinti. Visita di Grafi. Visita in Ampiezza. Visualizzazione della Visita. Sottografi.

Componenti Connesse. Distanze. Albero di visita. Generazione di Labirinti. Mappe come Grafi di Pixels. Visita di Grafi di Pixel.

## 22. **Web Crawling**

Gestione degli Errori. Web Crawling. Scaricare una Pagina. Elenco dei Links.

# 1

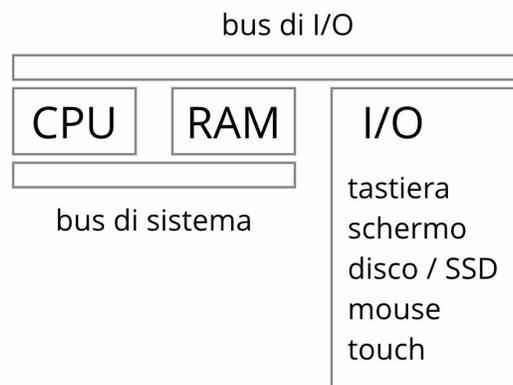
# I COMPUTER E LA PROGRAMMAZIONE

L'uso dei computer è ubiquo nel mondo moderno. I computer esistono in varie forme fisiche, come ad esempio desktops, laptops, tablets e smartphones. I computer sono macchine polivalenti in grado di eseguire applicazioni molte diverse. Possono ad esempio essere usati per scrivere testo, navigare il web, scrivere ed leggere emails o vedere filmati. Questo è possibile perché i computer sono programmabili. Per eseguire un'applicazione il computer esegue una serie di istruzioni contenute in un *programma*. La *programmazione* è l'attività di scrivere programmi per il computer.

Prima di iniziare a parlare concretamente di programmazione facciamo una breve panoramica sull'organizzazione dell'hardware, la rappresentazione delle informazioni in un computer e una brevissima storia dei linguaggi di programmazione.

## 1.1 L'ARCHITETTURA DEGLI ELABORATORI

L'hardware è la realizzazione fisica di una computer, cioè i circuiti integrati che lo definiscono. L'organizzazione di un computer, da un punto di vista logico-funzionale, non è cambiata molto da quando è stata proposta con sorprendente chiarezza da John von Neumann nel 1945 e che è conosciuta con il nome appunto di "architettura di von Neumann" e mostrata schematicamente qui a seguito.



Un computer è costituito dal processore centrale, o *Central Processing Unit* (CPU), dalla memoria centrale, o *Random Access Memory* (RAM), e da una serie di dispositivi di I/O, collegati da uno o più linee di comunicazione, o *bus*. La CPU esegue i programmi e tramite essi governa e controlla l'hardware dell'intero computer. I programmi e i dati che i programmi necessitano per l'elaborazione sono mantenuti nella memoria RAM. La CPU legge e scrive direttamente la memoria centrale. RAM è il nome generico di un tipo di tecnologia per le memorie. La memoria centrale comprende in realtà una gerarchia di memorie che oltre alla RAM vera e propria include memorie più piccole e veloci dette memorie *cache*. La CPU e la RAM sono connesse tramite una linea di comunicazione ad altissima velocità denominata bus di sistema.

La RAM è una memoria volatile, cioè mantiene il suo contenuto solamente mentre il computer è in funzione. Il disco o le memorie Flash sono memorie

permanenti, dette memorie secondarie, che mantengono invece il loro contenuto anche quando il computer è spento. I contenuti della RAM possono all'occorrenza essere scritti su disco e viceversa i contenuti del disco possono essere caricati in RAM.

Per interagire con l'esterno il computer usa una serie disparata di dispositivi di Input e Output (I/O), come tastiere, mouse, schermi e touch screens. Dal punto di vista architetturale, le memorie secondarie vengono considerate dispositivi di I/O. I dispositivi di I/O funzionano attraverso controllori hardware appositi che convertono eventi del mondo reale in informazioni per la CPU. In questo modo la CPU può essere avvertita di quando, ad esempio, il mouse è mosso o un tasto della tastiera è stato premuto. Viceversa, la CPU può inviare comandi ai dispositivi di output, come ad esempio definire il colore dei punti dello schermo. I dispositivi di I/O sono connessi al bus di sistema tramite un altro tipo di bus detto bus di I/O.

## 1.2 LA MEMORIA E I DATI

L'hardware è in grado di interpretare un solo tipo di dato elementare che è il bit, o *binary digit*, capace di assumere due stati **0** e **1**. Tutti i dati in un computer sono rappresentati tramite opportune sequenze di bit, ovvero sequenze di **0** e **1**, attraverso specifici formati. Ad esempio esistono formati per numeri, testo, immagini, musica, video, ecc. Qualsiasi cosa che può essere rappresentata da sequenze di bits può essere elaborata da un computer o trasmessa tramite reti di computer come Internet.

Per misurare la quantità di memoria si preferisce usare il byte che corrisponde a 8 bit consecutivi. Un byte può assumere 256 stati, cioè tutti i possibili stati di una sequenza di 8 bit. Un byte può rappresentare ad esempio tutti valori interi da 0 a 255.

La memoria centrale può essere vista, da un punto di vista funzionale, come una lunga sequenza di byte. Ogni byte della memoria è accessibile tramite il suo *indirizzo*, il primo byte ha indirizzo 0, il secondo 1, il terzo 2 e così via. Quindi la CPU, ovvero un programma, può leggere o scrivere un qualsiasi byte della memoria indicandolo con l'appropriato indirizzo.

Oggigiorno le memorie sono grandi come anche i dati da elaborare. Per misurarle si usano vari multipli del byte: il *kilobyte* (KB) corrispondente a 1024 byte, il *megabyte* (MB) corrispondente a 1024 KB, il *gigabyte* (GB) corrispondente a 1024 MB, e il *terabyte* (TB) corrispondente a 1024 GB. Tipiche memorie RAM per laptop e desktop si aggirano dai 4 ai 16 GB, mentre per gli smartphones vanno da 1 a 4 GB. Tipiche memorie secondarie vanno da 256 GB a 2 TB per laptop e desktop, e dai 16 ai 128 GB per gli smartphones.

Per rappresentare informazioni del mondo reale in binario occorre definire delle codifiche, che sono convenzioni su come convertire dati reali in sequenze di bits. Ad esempio, la tabella ASCII mette in relazione ogni carattere usato dagli alfabeti inglesi con un valore specifico rappresentabile

da un byte. Qui sotto riproduciamo una tabella ASCII da un manuale di stampante del 1972.

**USASCII code chart**

Bits												
b <sub>7</sub> b <sub>6</sub> b <sub>5</sub>												
b <sub>4</sub>	b <sub>3</sub>	b <sub>2</sub>	b <sub>1</sub>	Column	0 0 0	0 0 1	0 1 0	0 1 1	1 0 0	1 0 1	1 1 0	1 1 1
Row	0	1	2	3	4	5	6	7				
0 0 0 0	0	0	0	0	NUL	DLE	SP	0	@	P	\	p
0 0 0 1	0	0	1	1	SOH	DC1	!	1	A	Q	a	q
0 0 1 0	0	0	1	0	STX	DC2	"	2	B	R	b	r
0 0 1 1	0	0	1	1	ETX	DC3	#	3	C	S	c	s
0 1 0 0	0	1	0	0	EOT	DC4	\$	4	D	T	d	t
0 1 0 1	0	1	0	1	ENO	NAK	%	5	E	U	e	u
0 1 1 0	0	1	1	0	ACK	SYN	&	6	F	V	f	v
0 1 1 1	0	1	1	1	BEL	ETB	'	7	G	W	g	w
1 0 0 0	1	0	0	0	BS	CAN	(	8	H	X	h	x
1 0 0 1	1	0	0	1	HT	EM	)	9	I	Y	i	y
1 0 1 0	1	0	1	0	LF	SUB	*	:	J	Z	j	z
1 0 1 1	1	0	1	1	VT	ESC	+	:	K	[	k	{
1 1 0 0	1	1	0	0	FF	FS	,	<	L	\	l	
1 1 0 1	1	1	0	1	CR	GS	-	=	M	]	m	}
1 1 1 0	1	1	1	0	SO	RS	.	>	N	^	n	~
1 1 1 1	1	1	1	1	SI	US	/	?	O	_	o	DEL

La codifica di altri dati tipicamente necessita di più di un byte. Ad esempio i numeri usano 4 o 8 bytes. Per ogni tipo di dato possono esistere più di una codifica. Ad esempio, la codifica Unicode UTF8 estende l'ASCII per rappresentare i caratteri di tutti i possibili alfabeti. Un altro esempio sono le immagini che possono essere codificate come JPG o PNG.

## 1.3 LA CPU E I PROGRAMMI

Il comportamento di un computer durante l'esecuzione di una applicazione è controllato da un programma, che è definito da una serie di *istruzioni*. Le istruzioni sono contenute in memoria e codificate con opportune sequenze di bytes. La CPU è solo in grado di eseguire istruzioni molto semplici, come ad esempio:

- lettura e scrittura di una sequenza di bytes in memoria, tipicamente 4 o 8;
- lettura e scrittura di registri, che sono piccolissime memorie interne alle CPU;
- operazioni aritmetiche e logiche con operatori memorizzati nei registri;
- salti dell'esecuzione di istruzioni in un particolare punto del programma.

Una CPU esegue un semplice ciclo ripetutamente. Legge la prossima istruzione dalla memoria, la decodifica e la esegue leggendo dati dalla memoria, effettuando qualche operazione aritmetica o logica e scrivendo il risultato in memoria o in un registro. Poi ripete questo ciclo leggendo un'altra istruzione, ecc...

I processori sono velocissimi, potendo eseguire miliardi di istruzioni al secondo. Generalmente la velocità di un processore, cioè il numero di istruzioni che può eseguire al secondo, può essere approssimata dal numero di cicli che l'hardware esegue al secondo. L'unità di misura è l'*hertz* che corrisponde ad un ciclo al secondo, e i suoi multipli: il *kilohertz* (KHz) corrispondente a 1000 hertz, il *megahertz* (MHz) corrispondente a 1000 KHz, e il *gigahertz* (GHz) corrispondente a 1000 MHz. Le CPU attuali di notebooks, desktops e smartphones si aggirano da 1 a 3 GHz.

## 1.4 LINGUAGGI A BASSO LIVELLO

Ogni tipo di CPU ha il suo repertorio di istruzioni e questo è chiamato *linguaggio macchina*. Ad esempio, il linguaggio macchina di un processore x86, usato in notebooks e laptops, è differente da quello di un processore ARM, utilizzato in smartphones. In linea di principio tutti i computer potrebbe essere programmati direttamente nel loro linguaggio macchina.

I primissimi computer potevano essere programmati solamente in questo modo. Nonostante i programmi di allora non fossero molto grandi si sentì subito l'esigenza di poter programmare in un linguaggio più leggibile e quindi meno suscettibile a semplici errori di scrittura. Già negli anni '50 fu introdotto il *linguaggio assembly*. Questo linguaggio permette di scrivere le istruzioni in un modo simbolico più facile per i programmatori da comprendere.

La traduzione da assembly a linguaggio macchina è essere eseguita da un opportuno programma chiamato *assembler*. L'assembler è un primo esempio di un programma che manipola un altro programma. Questa idea è al cuore dello sviluppo software moderno.

Il linguaggio assembly è sicuramente più agevole del linguaggio macchina ma è ancora un linguaggio molto difficile da usare per scrivere programmi medio-grandi. Inoltre, al pari del linguaggio macchina è specifico per un certo tipo di processore. Un programma scritto nell'assembly di una particolare CPU, diciamo un processore x86 di un laptop, sarà differente da un programma per lo stesso compito per una differente CPU, ad esempio il processore ARM di un cellulare. Se si vuole convertire un programma in assembly per un processore in uno in assembly di un altro processore, il programma deve essere completamente riscritto. Per queste ragioni sono stati molto presto introdotti i cosiddetti linguaggi ad alto livello.

## 1.5 LINGUAGGI AD ALTO LIVELLO

*Software* è un termine generale per indicare le sequenze di istruzioni che determinano il comportamento di un computer. Ovvero, il software sono i programmi. Questi non sono più scritti direttamente in linguaggio macchina o in linguaggio assembly, eccetto casi molto rari, ma in linguaggi più facili e più leggibili per i programmatori, detti linguaggi ad alto livello. Ci penserà il computer stesso, tramite opportuni programmi, a tradurre in linguaggio macchina i programmi scritti in un linguaggio ad alto livello.

Tra la fine degli anni '50 e l'inizio degli anni '60 fu intrapreso uno dei più importanti passi nella storia della programmazione dei computer: lo sviluppo di un *linguaggio ad alto livello*, cioè un linguaggio indipendente dalla specifica architettura della CPU. Oltre a questo vantaggio, i linguaggi ad alto livello sono molto più vicini al modo di esprimersi degli esseri umani, permettendo di scrivere software più agevolmente.

Un programma scritto in un linguaggio ad alto livello è convertito da un programma traduttore nel linguaggio macchina di uno specifico processore. Se la traduzione avviene una sola volta, il linguaggio si dice *compilato* e il programma traduttore è un *compilatore*. Se la traduzione avviene ogni volta che il programma viene eseguito, il linguaggio si dice *interpretato* e il programma traduttore è un *interprete*. L'uso degli interpreti semplifica la creazione dei linguaggi di programmazione dato che, in generale, gli interpreti sono più semplici da scrivere dei compilatori, e possono essere portati da una architettura hardware all'altra molto semplicemente. I compilatori hanno invece il vantaggio che il programma eseguito è molto più veloce.

Esistono moltissimi linguaggi ad alto livello che differiscono per campo applicativo, metodo di traduzione, livello di astrazione dell'hardware. Sono stati inventati migliaia di linguaggi di programmazione ma solo poche centinaia sono usati. Tra i linguaggi compilati ci sono Java, C/C++, C#, Visual Basic. Tra quelli interpretati ci sono JavaScript, Python, PHP, Ruby.

Sebbene tutti i linguaggi di programmazione siano logicamente equivalenti, non sono affatto tutti ugualmente adatti per i vari tipi di programmi. C'è un mondo di differenza tra scrivere un programma in JavaScript che controlla una sofisticata pagina web e scrivere un programma in C++ che implementa il compilatore JavaScript.

## 1.6 STORIA DEI LINGUAGGI DI PROGRAMMAZIONE

I primi linguaggi ad alto livello erano dedicati a campi applicativi specifici. Alla fine degli anni '50, il FORTRAN, nome derivato da *Formula Translation*, fu sviluppato da un team dell'IBM per programmare calcoli scientifici e ingegneristici. Sempre negli anni '50 il COBOL, *Common Business Oriented Language*, si focalizzava su applicazioni gestionali e di contabilità. Sia il Fortran che il COBOL sono ancora usati. Il BASIC, *Beginner's All-purpose Symbolic Instruction Code*, sviluppato a Dartmouth nel 1964, fu concepito per l'insegnamento della programmazione. Per la sua semplicità fu il primo linguaggio ad alto livello dei personal computers. Programmi scritti in BASIC risultano molto più leggibile delle stesse versioni scritte in linguaggio assembly.

I linguaggi come FORTRAN, COBOL e BASIC, devono in parte il loro successo all'essere focalizzati su problemi specifici. Il loro design intenzionalmente non tentò di gestire la scrittura di ogni possibile applicazione. Durante gli anni '70 furono creati linguaggi per la programmazione di sistema, cioè per scrivere assembler, compilatori e sistemi operativi. Quello che ha avuto maggiore successo è il linguaggio C, sviluppato nel 1973, ed ancora oggi in uso. Negli anni '80 furono introdotti linguaggi come il C++ con l'intento di aiutare a gestire la complessità di programmi molto grandi. Il C++ si è evoluto a partire dal C ed è oggi uno dei linguaggi più usati. Ad esempio, la maggior parte del software su Mac è scritto in C, C++ e Objective-C, un dialetto del C, Word è scritto in C e C++, i sistemi operativi Unix e Linux sono scritti in C e i web browser Firefox e Chrome sono scritti in C++.

Durante gli anni '90 vari linguaggi furono sviluppati in congiunzione alla crescita di Internet e del World Wide Web. Lo scopo originale del linguaggio Java era la programmazione di piccoli sistemi incorporati in elettrodomestici e apparecchi elettronici. Ma invece diventò particolarmente utile come linguaggio per programmare servizi web. Quando si visita un sito come eBay, il nostro computer esegue il programma browser scritto in C++, ma i servers di eBay possono usare Java per preparare le pagine che sono inviate

al nostro browser.

A metà degli anni '90 fu creato JavaScript da Netscape per produrre effetti dinamici nei browser. Oggigiorno quasi tutte le pagine web includono codice JavaScript. JavaScript è un esempio di un linguaggio di *scripting*, che è flessibile e semplice da usare. Altri esempi di questo tipo nati negli anni '90 sono il Python per la programmazione generica, e il PHP per creare pagine Web. I linguaggi di scripting sono una categoria di linguaggi che permette di scrivere programmi che automatizzano l'esecuzione di compiti, eseguiti chiamando altri programmi o librerie. Una caratteristica i linguaggi di scripting è che non hanno bisogno di sfruttare al massimo la velocità del computer perché i compiti che sono soliti automatizzare demandano il calcolo intensivo ad altre componenti software. I programmi scritti in questi linguaggi sono spesso chiamati *scripts*.

Questo libro introduce i principianti alla programmazione usando il linguaggio *Python*, che è stato introdotto nel 1989 da Guido van Rossum per semplificare l'amministrazione di sistema. Il Python è un linguaggio di scripting interpretato. Intorno al 1991 la prima versione del linguaggio era pronta. Il nome *Python*, che Guido diede al suo linguaggio, deriva da *Monty Python's Flying Circus*, una serie televisiva comica britannica.

Originariamente Python venne subito usato con successo per l'amministrazione di sistema. Nel tempo Python è stato via via migliorato e oggi è un linguaggio collaudato e maturo. Python ha una libreria standard molto ricca ulteriormente estesa da una collezione di librerie esterne che facilitano lo sviluppo software in vari campi come web, grafica, giochi, multimedia, calcolo scientifico, ecc. Anche grazie a ciò l'uso di Python è andato crescendo in tutti gli ambiti commerciali, scientifici ed educativi. Inoltre, Python è un linguaggio molto facile da apprendere e da usare.

## 1.7 SVILUPPO SOFTWARE

Compilatori, web browsers e sistemi operativi sono esempi di programmi che possono avere decine di milioni di linee di codice. Per progettare, scrivere e mantenere tali sistemi occorrono centinaia di persone che lavorano insieme e la vita del sistema può durare decenni. Per avere un'idea di come il software è diventato sempre più complesso, si pensi che il sistema operativo Unix del 1975 consisteva in appena 9000 linee di codice C e fu scritto da due persone, Ken Thompson e Dennis Ritchie. Oggi Linux, una variante moderna di Unix, ha all'incirca 15 milioni di linee di codice.

Oggi chi si accinge a scrivere un programma difficilmente parte da zero. Come in una qualsiasi altra attività che ha a che fare con costruzioni complesse, anche la programmazione può usufruire di componenti già fatte e pronte all'uso. Per esempio, se si vuole scrivere un'applicazione per Windows, Mac, iPhone o Android, ci sono librerie di programmi già pronti per gli elementi dell'interfaccia grafica. La maggior parte del lavoro del programmatore odierno consiste nel capire quali componenti software sono disponibili e come combinarli insieme.

Molti componenti sono a loro volta complessi e sono costruiti a partire da librerie di componenti più semplici. Tutto ciò, a volte, per parecchi livelli. Per questo si può pensare allo sviluppo di software complesso come un lavoro astratto. Al di sotto di tutto, c'è il *sistema operativo* che è il programma che gestisce l'hardware, compresa l'esecuzione degli altri programmi. Il sistema operativo mette a disposizione dei servizi, chiamati *system calls*, che permettono ad esempio di accedere a file e manipolare la memoria. Lo strato successivo è costituito da librerie che forniscono servizi utili ai programmatori come il calcolo di funzioni matematiche, la crittografia, la grafica, la compressione di dati, ecc. Le applicazioni sono l'ultimo strato. Una tipica applicazione usa sia librerie che servizi del sistema operativo.

Gli ostacoli dello sviluppo software non si esauriscono con la complessità intrinseca nelle grandi dimensioni del software ma derivano anche dalla

natura stessa della programmazione che è stata ben descritta in un epigramma di Alan Perlis: "Programming is an unnatural act", la programmazione è un atto innaturale. La programmazione richiede un'attenzione ai dettagli che è difficile da mantenere. Inoltre, nessun programma abbastanza grande funziona la prima volta. Essenzialmente tutti i programmi contengono errori che produrranno comportamenti non voluti. Questi errori sono chiamati *bugs*, un termine inglese che significa piccolo insetto. Da qui anche il termine *debugging* per indicare l'attività, appunto, di trovare e correggere gli errori nei programmi. Infatti, i programmatori esperti passano più tempo a rimuovere gli errori che a scrivere i programmi stessi.

## 1.8 PROGRAMMAZIONE MODERNA

In questo libro, il nostro scopo è quello di scrivere programmi funzionanti che risolvono problemi reali in campi applicativi disparati. Non scriveremo sistemi complessi, ma programmi succinti che dimostrano le tecniche usate per risolvere i problemi presi in considerazione. Per farlo, faremo uso sia della estesa libreria standard inclusa con il linguaggio Python, sia di qualche libreria esterna per risolvere problemi specifici come ad esempio la creazione di interfacce. In questo senso, eviteremo di reinventare la ruota, riscrivendo funzionalità presenti in librerie comuni. Io credo che questo sia una buona introduzione alla programmazione moderna, in cui spesso la maggioranza dei programmatori deve comprendere come modellare problemi e come combinare librerie di componenti disparati per risolverli in modo efficiente ed efficace.

## 2

## PRIMI PASSI IN PYTHON

Python è un linguaggio interpretato. Per eseguire programmi in Python è necessario installare una distribuzione di Python che contiene l'interprete per eseguire i programmi, e la libreria standard che aiuta a risolvere una varietà di problemi disparati. Un programma Python non è altro che un testo scritto nel linguaggio stesso che viene eseguito dall'interprete. In questo primo capitolo introdurremo l'uso dell'interprete Python.

## 2.1 INSTALLAZIONE DI PYTHON

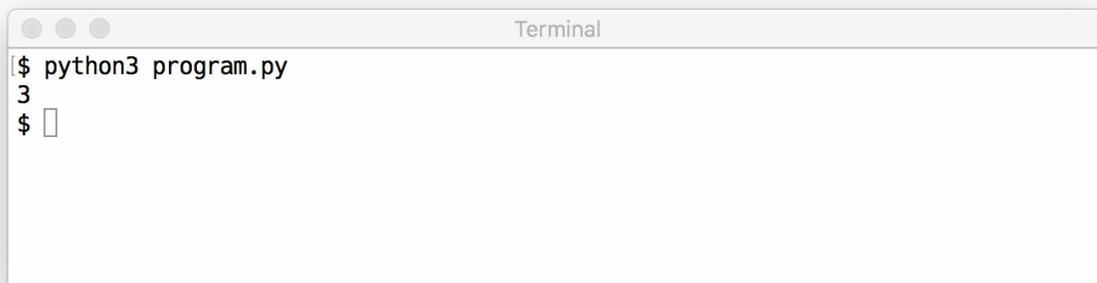
Python è un linguaggio in continua evoluzione, con varie versioni disponibili all'uso. Questo libro è stato scritto basandosi sulla versione 3.5 del linguaggio. Dato che le varie versioni sono incompatibili fra loro è necessario fare attenzione a scegliere la versione giusta. Per la maggior parte del libro, utilizzeremo librerie standard di Python o indicheremo dove scaricare le librerie necessarie. Per la parte di programmazione interattiva, utilizzeremo la libreria grafica Qt.

Il sito ufficiale di [Python](#) permette di scaricare la versione aggiornata del linguaggio. Per questo libro è consigliabile la distribuzione del linguaggio chiamata *Anaconda* che contiene Python, una collezione di librerie particolarmente utili nella programmazione interattiva e scientifica, e un interprete evoluto chiamato IPython o Jupyter. Anaconda è scaricabile dal sito di [Continuum Analytics](#), l'azienda che la supporta.

## 2.2 ESECUZIONE DI PROGRAMMI IN PYTHON

L'interprete Python viene eseguito da terminale. Il terminale è un'applicazione che permette di eseguire comandi di sistema. Per usare il terminale, basta lanciare l'applicazione `cmd.exe` su Windows, `Terminal.app` sul Mac e `Terminal` su Linux. I programmi Python si possono editare con qualunque editor di testo, come ad esempio l'editor gratuito multiplatforma [Atom](#).

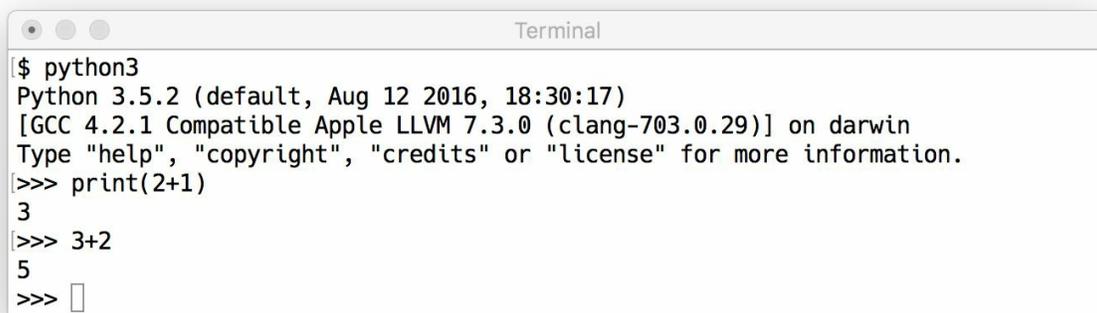
Un programma Python può essere eseguito da terminale invocando l'interprete Python `python3` con il nome del file che contiene il programma. Ad esempio, se salviamo il programma `print(2+1)` nel documento `program.py`, lo possiamo eseguire da terminale lanciando `python3 program.py`. Il programma stamperà il testo `3` sullo schermo come dimostrato di seguito.

A screenshot of a terminal window titled "Terminal". The terminal shows the command `python3 program.py` being executed, followed by the output `3` and a new prompt `$` with a cursor.

```
Terminal
[$ python3 program.py
3
$ ]
```

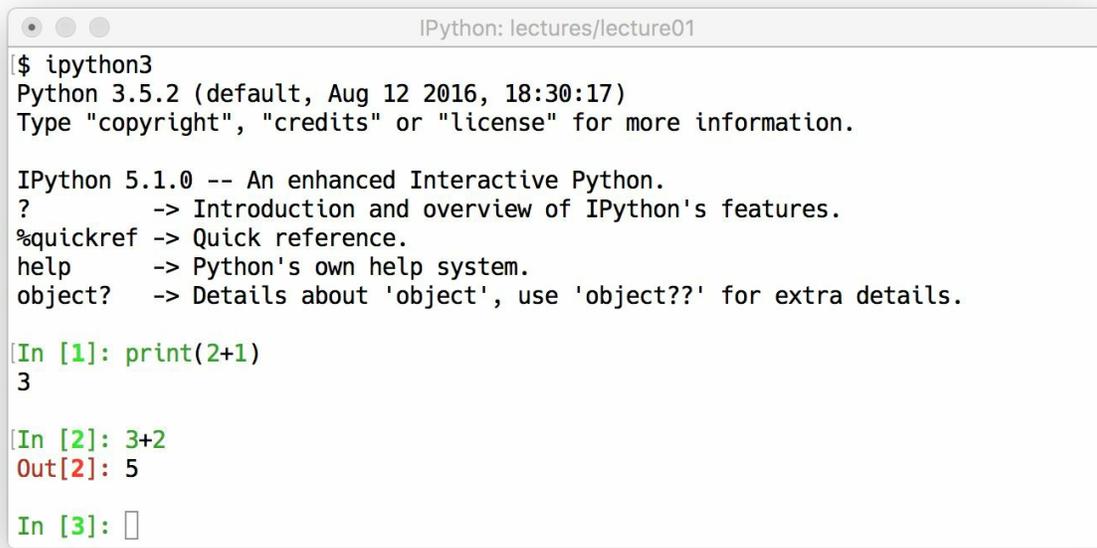
## 2.3 PROGRAMMAZIONE INTERATTIVA

Lo stesso programma può essere anche eseguito immettendo il codice riga per riga nell'interprete lanciato in modalità *shell*. Per invocare Python in questa modalità basta invocare l'interprete senza argomenti. Quando si invoca la shell, la stringa `>>>`, che si chiama *prompt*, indica che la shell è pronta a ricevere un comando. Per ogni comando inserito, la shell lo legge, ne calcola il valore o lo stampa. Nella shell non è necessario richiedere esplicitamente la stampa di un valore con `print()` come dimostrato di seguito.



```
Terminal
[$ python3
Python 3.5.2 (default, Aug 12 2016, 18:30:17)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> print(2+1)
3
[>>> 3+2
5
>>> ]
```

Il linguaggio Python è supportato da varie shell. IPython è una shell avanzata che supporta la colorazione del programma e permette di editarlo più facilmente. In IPython il prompt è del tipo `In [i]:`, dove `i` è il numero del comando, e i risultati sono stampati con prompt `Out[i]:`, come dimostrato a seguito.



```
IPython: lectures/lecture01
[$ ipython3
Python 3.5.2 (default, Aug 12 2016, 18:30:17)
Type "copyright", "credits" or "license" for more information.

IPython 5.1.0 -- An enhanced Interactive Python.
?      -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help   -> Python's own help system.
object? -> Details about 'object', use 'object??' for extra details.

[In [1]: print(2+1)
3

[In [2]: 3+2
Out[2]: 5

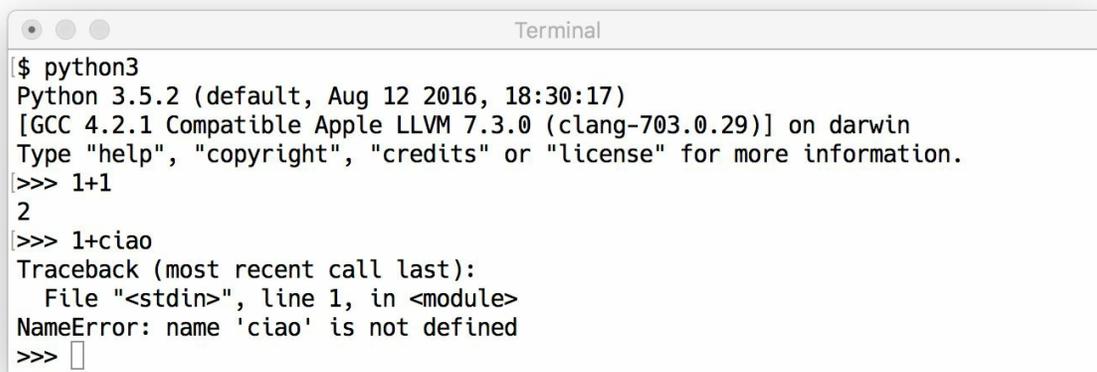
In [3]: ]
```

L'esecuzione da shell è particolarmente adatta a sviluppare codice in modo interattivo, che permette di sperimentare vari modi di risolvere un problema eseguendo pezzi di codice diversi su dati di test. Una volta che il programma è completo, lo si può salvare in un file per poterlo eseguire ripetutamente.

Una differenza fondamentale tra i due modi di esecuzione, da programma o da shell, è che la shell stampa tutti i valori risultanti dalle operazioni eseguite, mentre il programma eseguito da terminale stampa solo i valori esplicitamente indicati con `print()`. Questa differenza deriva dal fatto che la shell interattiva è ottimizzata per sviluppare programmi, mentre l'esecuzione da terminale è più adatta quando si usa un programma finito per manipolare dati.

## 2.4 ERRORI DI PROGRAMMAZIONE

Nello scrivere i vostri programmi, incontrerete spesso errori di programmazione. Infatti, la correzione degli errori, detta anche *debugging*, è la maggiore attività dei programmatori. Python è un linguaggio particolarmente utile perché cerca di indicare esplicitamente gli errori di programmazione. Ad esempio,



```
Terminal
[$ python3
Python 3.5.2 (default, Aug 12 2016, 18:30:17)
[GCC 4.2.1 Compatible Apple LLVM 7.3.0 (clang-703.0.29)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
[>>> 1+1
2
[>>> 1+ciao
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'ciao' is not defined
>>> ]
```

La prima istruzione `1+1` è andata a buon termine e eseguita con risultato 1. La seconda istruzione `1+ciao` ha causato un errore. In questo caso il programma non è valido e la linea `Name Error: name 'ciao' not defined` indica il tipo di errore. Non preoccupatevi ora di comprendere questo particolare errore. Ma è importante abituarsi a leggere i messaggi di errore di Python, dato che contengono informazioni utile per la correzione del programma stesso.

Gli errori più insidiosi sono quelli che Python non può trovare. Questi errori sono tali per cui il programma esegue in modo corretto, ma calcola valori non desiderati. Questi errori saranno molto più difficili da trovare e correggere.

## 2.5 RISORSE PER PROGRAMMARE IN PYTHON

Questo libro è stato scritto per principianti, e non assume alcuna conoscenza di concetti di programmazione. La risorsa più utile che suggeriamo di utilizzare nell'imparare a programmare in Python è la **documentazione** del linguaggio stesso. Infatti, in questo libro non ripeteremo informazioni dettagliate sulle funzioni standard di Python, dato che queste informazioni si possono trovare in modo più completo nella documentazione.

Un'altra risorsa utile per i principianti può essere il visualizzatore interattivo **Online Python Tutor** che permette di tracciare l'esecuzione dei programmi Python. Ci sono anche vari *Integrated Development Environment* (IDE) che sono editor avanzati che possono essere usati per creare programmi Python. Per quanto questi possano essere utili, suggerisco di iniziare a programmare senza questi strumenti avanzati, che diventano più utili per programmi larghi.

Infine, per quanto la libreria standard di Python contenga moltissime funzionalità, è a volte necessario installare pacchetti aggiuntivi che arricchiscono il linguaggio. Ad esempio, Anaconda contiene molti di questi pacchetti aggiuntivi. Per installarne ulteriori pacchetti consigliamo di usare la funzionalità `pip3 install <nome pacchetto>` o seguire le indicazioni degli autori delle librerie stesse.

# 3

## PYTHON COME CALCOLATRICE

Python, eseguito da shell, può essere usato come un calcolatrice avanzata. Come ogni calcolatrice, Python supporta tutte le operazioni aritmetiche, varie funzioni matematiche, ed ha la capacità di memorizzare valori parziali usando *variabili*. Per semplicità, nei capitoli qui di seguito, non riportiamo il prompt della shell in modo esplicito, ma riportiamo i valori stampati dalla shell, o dal programma eseguita in modo non-interattivo, preceduti da `# Out:`.

## 3.1 ESPRESSIONI ARITMETICHE

Python supporta le comuni operazioni aritmetiche: addizione `+`, sottrazione `-`, moltiplicazione `*`.

```
15 + 3
# Out: 18
10 - 25
# Out: -15
3*7
# Out: 21
```

Python supporta sia numeri interi che numeri con virgola, detti *floating point* o in virgola mobile. Se almeno uno degli operandi è un numero con virgola, l'operazione è automaticamente effettuata in virgola mobile.

```
15.7 + 3
# Out: 18.7
18.0 * 5
# Out: 90.0
```

L'eccezione alla regola precedente è la divisione. In Python esistono due tipi di divisione: la divisione standard `/`, che ritorna sempre un numero in virgola mobile, e la divisione intera `//` che ignora la parte frazionaria del quoziente.

```
15 // 2
# Out: 7
15.0 // 2
# Out: 7.0
15 / 2
# Out: 7.5
15.0 / 2
# Out: 7.5
```

Gli operatori `*` e `/` hanno la precedenza su `+` e `-`, così in `16 - 2*3` prima è effettuata la moltiplicazione e poi la sottrazione. Per raggruppare risultati parziali si possono usare le parentesi tonde.

```
(12/5)*(16 - 2*3)
# Out: 24.0
```

Python supporta anche operatori più avanzati come il resto della divisione `%` e l'elevamento a potenza `**`.

```
24 % 7
# Out: 3
27 % 9
# Out: 0
27 % 7.5
# Out: 4.5
2**8
# Out: 256
2**0.5
# Out: 1.4142135623730951
```

I risultati delle operazioni con interi hanno precisione illimitata mentre quelli in virgola mobile sono approssimazioni dai valori, avendo precisione limitata. Ad esempio,

```
2**100
# Out: 1267650600228229401496703205376
2.0**100
# Out: 1.2676506002282294e+30
```

## 3.2 COMMENTI

Si possono inserire commenti all'interno dei programmi. Questi servono per lasciare note e spiegazioni circa il funzionamento del codice, sia per se stessi che per altri. In Python, ogni riga che inizia col simbolo `#` è un commento e non viene eseguito dall'interprete. Abbiamo usato i commenti nel codice precedente per indicare i valori calcolati da Python con `# Out: .`

## 3.3 STAMPA DI VALORI

I valori calcolati in precedenza e indicati nelle linee `# Out:` sono solo visibili quando il programma è eseguito in modalità interattiva, cioè nella shell. Per forzare la stampa di valori, usiamo l'istruzione `print(espressione)`.

```
print(1+1)
# Out: 2
```

Per stampare più espressioni sulla stessa riga basta passare a `print()` una lista di espressioni separate da virgole.

```
print(2+3,5+6)
# Out: 5 11
```

## 3.4 STRINGHE

In Python, il testo si rappresenta come una sequenza di caratteri chiamata *stringa*. Per rappresentare una stringa basta racchiudere la sequenza di caratteri tra apici singoli `'` o doppi `"`. Usando i doppi apici possiamo usare all'interno della stringa gli apici singoli e viceversa. Si può indicare una stringa che non contiene nessun carattere, detta stringa vuota, non includendo nulla tra gli apici.

```
'ciao'  
# Out: 'ciao'  
print('ciao')  
# Out: ciao  
  
print("L'altra mattina")  
# Out: L'altra mattina  
  
# stringa vuota  
' '  
# Out: '  
print('')  
# Out:
```

Notare come `print()` non stampa gli apici perché questi sono solo una notazione usata in Python per indicare l'inizio e la fine del testo. Usando `repr()` invece otteniamo la rappresentazione di Python che è delimitata da apici, la stessa stampata dalla shell in automatico.

```
print(repr('ciao'))  
# Out: 'ciao'
```

Se il testo prende più linee, possiamo racchiuderlo tra tre apici singoli `'''` o tre doppi apici `"""` e possiamo anche usare liberamente singoli e doppi apici:

```
print('''Questo testo
va a capo.''' )
# Out: Questo testo
# Out: va a capo.

print(repr('''Questo testo
va a capo.'''))
# Out: 'Questo testo\nva a capo.'
```

Notare come la funzione `print()` riproduce le andate a capo correttamente, mentre usando `repr()` il valore della stringa contiene ora una codifica del carattere di fine linea `\n`. Questo carattere non è rappresentabile direttamente in una stringa quindi va indicato in modo speciale attraverso *sequenze di escape*. I caratteri che possono essere immessi in questo modo sono: `\n` per fine linea, `\'` per l'apice singolo, `\"` per il doppio apice, `\t` per il tab e `\\` per il carattere backslash stesso `\`.

In Python le stringhe possono rappresentare il testo di qualsiasi lingua nota. Ad esempio possiamo usare senza problemi le lettere accentate italiane.

```
print('né così né cosà')
# Out: né così né cosà
```

L'operatore `+` concatena due stringhe mentre l'operatore `*` ripete una stringa più volte.

```
saluto = 'Buon ' + 'giorno'
print(saluto)
# Out: Buon giorno
boom = 'tic tac '*5 + 'BOOM!'
print(boom)
# Out: tic tac tic tac tic tac tic tac tic tac BOOM!
```

## 3.5 TIPI E CONVERSIONI

Python distingue tra numeri interi, numeri in virgola mobile e stringhe associando ad ogni valore un tipo. I numeri interi sono di tipo `int`, quelli in virgola sono di tipo `float`, e le stringhe sono di tipo `str`. Parleremo dei tipi in dettaglio più avanti. Se si desidera conoscere il tipo di una espressione, basta usare la funzione `type(expr)`.

```
print(type(5 + 2))
# Out: <class 'int'>
print(type(10 + 5 / 2))
# Out: <class 'float'>
print(type('ciao'))
# Out: <class 'str'>
```

Notate come gli operatori `+` e `*` eseguono operazioni differenti a seconda del tipo degli operandi. Per i numeri, tipi `int` e `float`, queste sono le consuete operazioni aritmetiche. Per le stringhe gli operatori eseguono la concatenazione e la ripetizione. Nel gergo dei linguaggi di programmazione si dice che gli operatori sono *overloaded*. Se proviamo a usare l'operatore `+` con operandi misti abbiamo un errore.

```
print('stringa' + 5)
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: TypeError: Can't convert 'int' object to str implicitly
```

Se si vogliono convertire valori in tipi diversi, basta usare il nome del tipo passandogli il valore tra parentesi tonde. Per convertire un numero in una stringa basta chiamare `str()` con il valore numerico. Viceversa, se si vuole convertire una stringa in un numero, basta usare `int()` o `float()`.

```
# conversione in stringhe
print('stringa' + str(5))
```

```
# Out: stringa5

# conversione in numeri
print(10 + int('5'))
# Out: 15

# conversione non valida
print(int('ciao'))
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: ValueError: invalid literal for int() with base 10: 'ciao'
```

## 3.6 VARIABILI E ASSEGNAMENTI

Per facilitare la memorizzazione di valori e il loro successivo utilizzo, tutti i linguaggi di programmazione permettono di usare le variabili. Una *variabile* è un nome a cui è associato un valore. Il nome di una variabile può comprendere lettere, cifre e il carattere underscore `_`, ma non deve iniziare con una cifra. Una variabile è creata nel momento in cui gli è assegnato un valore con una istruzione di *assegnamento* che in Python ha la forma

```
nome_variable = espressione
```

Una istruzione di assegnamento prima calcola il valore dell'espressione a destra del segno `=`, e poi ne assegna il valore alla variabile il cui nome è a sinistra del segno `=`. Ad esempio, per creare una variabile di nome `piGRECO` ed assegnarle il valore `3.14` basta eseguire l'istruzione

```
piGRECO = 3.14
```

Per utilizzare il valore di una variabile in una espressione, basta indicarne il nome nell'espressione.

```
# area di un cerchio di raggio 10
raggio = 10
area = piGRECO * raggio ** 2
print(area)
# Out: 314.0
```

Le variabili sono chiamate così perché il valore assegnato può essere cambiato, assegnando alla variabile un nuovo valore. Per questo in programmazione il significato dell'assegnamento è diverso dall'uguaglianza in matematica, anche se si usa lo stesso simbolo `=` per indicare entrambi.

```
# ricalcolo dell'area con raggio 20
```

```
raggio = 20
area = pigreco * raggio ** 2
print(area)
# Out: 1256.0
```

In Python, si possono assegnare valori a qualunque variabile, ma solo leggere valori da variabili già assegnate. Se tentiamo di leggere il valore da una variabile non ancora definita abbiamo un errore.

```
print(2 * non_definita)
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: NameError: name 'non_definita' is not defined
```

L'errore è avvenuto alla linea 1 del file `<input>`, che indica la shell. L'ultima linea del messaggio, `NameError: name 'non_definita' is not defined`, indica che si è tentato di leggere il nome `'raggio'` che non è ancora definito.

Le variabili in Python possono contenere qualunque tipo di dati, come ad esempio numeri e stringhe.

```
saluto = "ciao"
print(repr(saluto))
# Out: 'ciao'
print(saluto)
# Out: ciao

stringa_vuota = ''
print(stringa_vuota)
# Out:
```

In Python, una variabile è solo un nome usato per riferirsi ad un valore. Per questo possiamo cambiare il tipo di dato a cui una variabile si riferisce facendo ulteriori assegnamenti. La ragione è che il tipo dei dati in Python è associato ai valori, non alle variabili, che sono solo nomi.

```
variabile = 10
print(type(variabile))
```

```
# Out: <class 'int'>

variabile = 'ciao'
print(type(variabile))
# Out: <class 'str'>
```

Per quanto questa funzionalità sia utile ai programmatori esperti, cambiare il tipo a cui una variabile di riferisce è sconsigliabile per i principianti perché può introdurre errori difficili da trovare successivamente.

## 3.7 INCREMENTI

È a volte necessario incrementare il valore di una variabile a partire dal valore che le è già stato assegnato. Per fare ciò possiamo semplicemente assegnare ad una variabile una espressione che la contiene. Questa funziona perché il valore dell'espressione è calcolato per primo, e poi viene assegnato nuovamente alla variabile.

```
# raddoppia il raggio  
print(raggio)  
# Out: 20  
raggio = raggio * 2  
print(raggio)  
# Out: 40
```

Questo tipo di assegnamento è talmente comune che Python predispone operatori speciali di incremento. Questi operatori sono del tipo `variabile <op>= espressione` e corrispondono ad assegnamenti del tipo `variabile = variabile <op> espressione`. Gli operatori supportati sono: `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, ecc.

```
# raddoppia l'altezza  
print(raggio)  
# Out: 40  
raggio *= 2  
print(raggio)  
# Out: 80
```

## 4 RIUTILIZZO DI ISTRUZIONI

Nel capitolo precedente abbiamo introdotto l'uso di Python per il calcolo di valori. Ritornando al piccolo programma che abbiamo scritto per calcolare l'area del cerchio, se volessimo calcolare l'area con un differente raggio dovremmo riscrivere le istruzioni ogni volta. Questo non solo è tedioso, ma una sorgente di possibili di errori. Python, come tutti i linguaggi di programmazione, mette a disposizione una varietà di costrutti per riutilizzare istruzioni.

## 4.1 FUNZIONI

Come le variabili permettono di riusare valori dandogli nomi, le *funzioni* permettono di dare un nome a sequenze di istruzioni. In Python, una funzione è definita con la sintassi

```
def nome_funzione(lista_parametri):  
    lista_istruzioni  
    return espressione # opzionale
```

La parola chiave `def` introduce la definizione della funzione `nome_funzione`. A questo nome viene assegnata la lista di istruzioni `lista_istruzioni`. Dopo il nome, tra parentesi, ci sono zero, uno o più parametri. Un *parametro* è una variabile a cui è assegnato un valore quando la funzione viene eseguita. Quindi i parametri permettono di fornire gli input alle istruzioni specificate nella funzione. Dopo i parametri ci sono i due punti `:` che indicano l'inizio della lista di istruzioni da eseguite ad ogni chiamata della funzione. Queste sono specificate nelle righe successive e *indentate di 4 spazi*. L'output della funzione è prodotto dall'istruzione `return espressione` che valuta l'espressione e poi termina l'esecuzione della funzione ritornando il valore dell'espressione a chi esegue la funzione.

Per *chiamare* la funzione, cioè eseguirne le istruzioni, basta utilizzare il nome della funzione in qualsiasi espressione, indicando tra parentesi i valori delle variabili da passare. La sintassi per la chiamata è quindi

```
nome_funzione(lista_valori)
```

Ad esempio, per calcolare l'area di un cilindro per diversi raggi e altezze, basta definire la funzione `area_cilindro` e poi chiamarla con argomenti diversi.

```
def area_cilindro(raggio, altezza):  
    pigreco = 3.14159
```

```
area = pigreco * raggio ** 2
circonferenza = 2 * pigreco * raggio
return 2 * area + altezza * circonferenza

print(area_cilindro(10, 5))
# Out: 942.477
print(area_cilindro(20, 10))
# Out: 3769.908
```

In questo esempio, la chiamata `area_cilindro(10, 5)` assegna il valore `10` al parametro `raggio` e `5` a `altezza` immediatamente prima di eseguire le istruzioni della funzione.

Tutte le variabili create in una funzione, compresi i parametri, sono *variabili locali* alla funzione, cioè non esistono al di fuori della funzione. Ad esempio, se cerchiamo di accedere al valore della variabile `circonferenza` otteniamo un errore.

```
print(circonferenza)
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: NameError: name 'circonferenza' is not defined
```

Le variabili create al di fuori di ogni funzione sono dette *variabili globali* e sono accessibili in lettura da qualunque funzione. Per quanto questo sia valido, è in generale sconsigliato accedere a variabili globali nelle funzioni dato che questi accessi sono spesso causa di errori di programmazione molto difficile da individuare.

## 4.2 FUNZIONI PREDEFINITE

Python ha molte funzioni già disponibili, dette predefinite o *builtin*. Si consulti la documentazione di Python per un elenco completo. Abbiamo già visto esempi di funzioni predefinite come `print()`, `repr()` e `type()`. In questo libro, introdurremo l'uso di altre funzioni predefinite quando necessario per risolvere un problema. Per il calcolo numerico sono ad esempio utili le funzioni per il valore assoluto `abs()` e per l'arrotondamento di numeri frazionari `round()`.

```
print(abs(-5))
# Out: 5
print(abs(-3.6))
# Out: 3.6
print(round(5.8))
# Out: 6
print(round(5.3))
# Out: 5
```

## 4.3 FUNZIONI CHE RITORNANO PIÙ VALORI

Come ulteriore esempio di funzione scriviamo le istruzioni per una funzione che prende in input un numero di secondi e ritorna l'equivalente in ore, minuti e secondi. Per ritornare più valori basta includere una lista di espressioni separate da virgole nell'istruzione `return`. Per ora possiamo solo stampare questi valori. Nei capitoli seguenti vedremo come accedere ai singoli valori di ritorno.

```
def hms(nsec):  
    hh = nsec // 3600  
    nsec = nsec % 3600  
    mm = nsec // 60  
    ss = nsec % 60  
    return hh, mm, ss
```

```
print(hms(4000))  
# Out: (1, 6, 40)  
print(hms(100000))  
# Out: (27, 46, 40)
```

## 4.4 PARAMETRI OPZIONALI

Finora abbiamo considerato solamente il modo basilare di definire i parametri di una funzione, dove ad ogni parametro deve essere assegnato un valore quando la funzione viene chiamata. A volte una funzione ha uno o più parametri a cui potrebbero essere assegnati dei ragionevoli valori di default liberando così il chiamante dal dover specificare esplicitamente tutti i valori.

Ad esempio, potremmo aggiungere valori di default alla funzione `area_cilindro()` definendo il valore 1 come default per raggio e altezza. Per definire un valore di default basta aggiungere `= valore` dopo il nome del parametro. Nel gergo di Python un parametro con valore di default è chiamato *parametro di default* o *parametro opzionale*.

```
def area_cilindro(raggio=1,altezza=1):
    pigreco = 3.14159
    area = pigreco * raggio ** 2
    circonferenza = 2 * pigreco * raggio
    return 2 * area + altezza * circonferenza
```

La funzione così definita si può chiamare con 0, 1, o 2 argomenti.

```
# equivalente a area_cilindro(1,1)
print(area_cilindro())
# Out: 12.56636

# equivalente a area_cilindro(2,1)
print(area_cilindro(2))
# Out: 37.699079999999995

# equivalente a area_cilindro(2,3)
print(area_cilindro(2,3))
# Out: 62.831799999999994
```

Quando una funzione ha molti argomenti e questi sono di default, è conveniente poter chiamare la funzione specificando il valore di un argomento non tramite la sua posizione ma tramite il nome del parametro. Per farlo usiamo la sintassi `parametro=valore` nella chiamata. Un argomento così specificato è chiamato *argomento chiave*. Gli argomenti chiave devono stare tutti in fondo alla lista degli argomenti, cioè non ci può essere un argomento chiave seguito da un argomento non chiave, e non si può specificare il valore di un argomento due volte.

```
# equivalente a area_cilindro(1,2)
print(area_cilindro(altezza=2))
# Out: 18.849539999999998
```

Python usa parametri di default in modo molto esteso nella sua libreria. Ad esempio, la funzione `round()` ha un parametro opzionale che definisce il numero di cifre decimali da mantenere, di default 0.

```
print(round(3.125))
# Out: 3
print(round(3.125,2))
# Out: 3.12
```

## 4.5 MODULI E FILE

Per riutilizzare variabili e funzioni in programmi diversi, Python permette di importare il contenuto di un file in un altro, o nella shell stessa. Ad esempio, se il file `modulo.py` contiene le funzioni

```
def area_sfera(raggio):
    return 4 * 3.14 * raggio ** 2

def volume_sfera(raggio):
    return 4/3 * 3.14 * raggio ** 3
```

possiamo accedere a tali funzioni importando il file con `import nome_modulo` e chiamando le funzioni con `nome_modulo.nome_funzione`.

```
import modulo

print(modulo.area_sfera(10))
# Out: 1256.0
print(modulo.volume_sfera(10))
# Out: 4186.666666666667
```

In generale, un modulo è un qualsiasi programma in Python salvato in un singolo file. Di solito, le funzioni e le variabili in un modulo sono in qualche modo connesse tra loro, ad esempio potrebbero essere funzioni matematiche o funzioni per la manipolazione dei file o per la grafica, ecc. L'aspetto importante è che le funzioni di un modulo possono essere usate in un qualsiasi programma senza doverle riscrivere.

Per chiamare le funzioni di un modulo evitando di menzionare il nome del modulo, basta usare la sintassi `from nome_modulo import nome_funzione`, per una funzione specifica, e `from nome_modulo import *` per tutte le funzioni. In generale, è consigliabile usare la versione del comando non abbreviata per ridurre i possibili errori di programmazione.

```
from modulo import area_sfera
print(area_sfera(10))
# Out: 1256.0
```

```
from modulo import *
print(volume_sfera(10))
# Out: 4186.666666666667
```

La libreria standard di Python comprende tantissimi moduli fra cui il modulo `math` per le funzioni matematiche.

```
import math

# logaritmo
print(math.log(10))
# Out: 2.302585092994046

# pi greco
print(math.pi)
# Out: 3.141592653589793

# fattoriale
print(math.factorial(20))
# Out: 2432902008176640000
```

Quando si fa l'import di un modulo l'interprete deve sapere dove cercare il corrispondente file. Nel caso dei moduli della libreria standard come `math` non ci sono problemi perché si trovano in una directory pre-impostata. Nel caso di moduli esterni, questi devono essere presenti nella stessa cartella da cui è stata invocata la shell o il programma attuale.

## 4.6 DOCUMENTAZIONE INTERATTIVA

Per avere un elenco delle funzioni in un modulo si può usare la funzione `dir()`:

```
print(dir(math))
# Out: ['__doc__', '__file__', '__loader__', '__name__',
'__package__', '__spec__', 'acos', 'acosh', 'asin', 'asinh',
'atan', 'atan2', 'atanh', 'ceil', 'copysign', 'cos', 'cosh',
'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs', 'factorial',
'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot', 'inf',
'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians',
'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'trunc']
```

Inoltre, possiamo avere la documentazione di ogni funzione di un modulo, dopo averlo importato, con la funzione `help()`. La funzione `help()` può anche essere usata fornendogli in input il nome di un modulo e in questo caso mostra la documentazione dell'intero modulo, cioè di tutte le funzioni contenute in esso.

```
help(math.log)
# Out: Help on built-in function log in module math:
# Out:
# Out: log(...)
# Out:     log(x[, base])
# Out:
# Out:     Return the logarithm of x to the given base.
# Out:     If the base not specified, returns the natural logarithm
(base e) of x.
# Out:
```

La funziona `help()` deriva le descrizione dal codice Python stesso. La descrizione di una funzione si può includere con un testo incluso tra tripli apici, singoli o doppi, subito dopo il nome della funzione. Questo testo si

chiama *docstring*. È consigliabile scrivere sempre una breve spiegazione di cosa fa una funzione e quale è il significato dei parametri di input.

```
def cubo(x):  
    '''Calcola il cubo di un numero.'''  
    return x ** 3  
  
print(cubo(5))  
# Out: 125  
  
help(cubo)  
# Out: Help on function cubo:  
# Out:  
# Out: cubo(x)  
# Out:     Calcola il cubo di un numero.  
# Out:
```

## 5 PRENDERE DECISIONI

Per implementare programmi complessi è necessario *prendere decisioni* su quali istruzioni eseguire in base ai dati in ingresso. Una decisione è presa in dipendenza del valore di una certa *condizione*. Ad esempio, per decidere se un numero è pari o dispari la condizione è se il resto della divisione per 2 è uguale a 0 o a 1. La condizione può essere semplice o complessa ma il suo valore sarà sempre o vero o falso.

## 5.1 VALORI BOOLEANI

Valori che possono essere solo veri o falsi sono detti *booleani*, in Python rappresentati col tipo `bool`. Un valore booleano può essere solo `True` o `False`, *vero* o *falso*.

```
booleano = True
print(booleano)
# Out: True
print(type(booleano))
# Out: <class 'bool'>
```

Sui valori booleani sono definiti tre *operatori logici*. L'operatore `not` nega un valore booleano, l'operatore `and` è vero solo se entrambi i suoi operatori sono veri, e l'operatore `or` è vero quando almeno uno dei suoi operandi è vero. Più formalmente `not x` è `True` se e solo se `x` è `False`, `x and y` è `True` se e solo se `x` e `y` sono `True` e `x or y` è `True` se e solo se o `x` è `True` o `y` è `True` o entrambi sono `True`. Si osservi che mentre gli operatori `not` e `and` hanno sostanzialmente lo stesso significato che hanno nel linguaggio comune, questo non è così per l'operatore `or`. Infatti, se ad esempio si dice "Mario ha le scarpe nere o la maglietta blu", l'interpretazione comune è che o è vera la prima possibilità o è vera la seconda ma non entrambe. Questa interpretazione si chiama *or esclusivo* mentre l'interpretazione nella logica booleana è detta *or inclusivo* perché risulta vero anche quando sono entrambe vere. Vediamo qualche esempio.

```
# operazioni semplici
freddo = True
caldo = not freddo
print(caldo)
# Out: False

pioggia = False
nuvoloso = True
brutto_tempo = pioggia or nuvoloso
```

```
print(brutto_tempo)
# Out: True

vento = True
neve = True
tormenta = vento and neve
print(tormenta)
# Out: True

# combinazione di operazione semplici
bel_tempo = not pioggia and not nuvoloso and not neve
print(bel_tempo)
# Out: False

# una espressione equivalente
bel_tempo = not (pioggia or nuvoloso or neve)
print(bel_tempo)
# Out: False
```

## 5.2 OPERATORI RELAZIONALI

Gli *operatori relazionali* confrontano valori permettendo di determinare se sono uguali o se uno è maggiore o minore di un altro. Il risultato di un operatore relazione è un valore booleano. Gli operatori relazionali in Python sono i seguenti: minore `<`, maggiore `>`, minore o uguale `<=`, maggiore o uguale `>=`, uguale `==` e diverso `!=` o `<>`. So noti come l'uguaglianza non è indicata dal simbolo matematico `=` per evitare di essere confuso con l'assegnamento che usa quel simbolo.

```
# confronti semplici
print(3 < 5)
# Out: True
print(10 == 11)
# Out: False
print(10.0 == 10)
# Out: True
print(3.5 != 3.45)
# Out: True

# combinazione di confronti su espressione
x = 2
y = 5
z = 4
print("valori ordinati:", x <= y and y <= z)
# Out: valori ordinati: False
# versione compatto del doppio confronto
print("valori ordinati:", x <= y <= z)
# Out: valori ordinati: False
```

Le stringhe sono confrontate secondo l'ordinamento lessicale, cioè confrontando i singoli caratteri da sinistra a destra e differenziando maiuscole e minuscole. Se tutti i caratteri sono uguali le due stringhe sono uguali; se una è il prefisso dell'altra quella più corta è la minore; altrimenti il primo carattere in cui le stringhe differiscono determina l'ordine.

```
print('Mario' == 'Bruno')
# Out: False
print('Mario' > 'Bruno')
# Out: True
print('Ma' < 'Mario')
# Out: True

print('Stringa' < 'stringa')
# Out: True
print("stringa" == "Stringa")
# Out: False
```

## 5.3 OPERATORE DI APPARTENENZA

Nelle stringhe l'operatore `in` permette di testare l'appartenenza di una sottostringa nella stringa originale. Il suo opposto è l'operatore `not in`.

```
esclamazione = 'Che bel tempo!'
print('C' in esclamazione)
# Out: True
print('c' in esclamazione)
# Out: False
print(' ' in esclamazione)
# Out: True
print('bel' in esclamazione)
# Out: True
print('ciao' not in esclamazione)
# Out: True
```

## 5.4 ISTRUZIONI CONDIZIONALI

In Python il costrutto che permette di prendere decisioni è l'istruzione `if` che nella sua forma base ha la sintassi

```
if condizione:
    istruzioni
```

`if` è la parola chiave che inizia l'istruzione, a seguire c'è una `condizione` e dopo i due punti `:` ci sono le `istruzioni`, indentate di 4 spazi, che saranno eseguite solo se il valore della `condizione` è vera ossia `True`.

```
pioggia = False
nuvoloso = True
if pioggia or nuvoloso:
    print("usciamo con l'ombrello")
# Out: usciamo con l'ombrello

nuvoloso = False
if pioggia or nuvoloso:
    print("usciamo con l'ombrello")
# Non stampa nulla dato che la condizione è False
```

Si possono anche specificare istruzioni diverse che vengono eseguite quando la condizione è `False` seguendo l'`if` con l'istruzione `else:` e il relativo blocco `istruzioni_else`.

```
if condizione:
    istruzioni
else:
    istruzioni_else
```

In questo caso, il blocco `istruzioni` viene eseguito se la condizione è vera, altrimenti si eseguono le istruzioni in `istruzioni_else`. Le istruzioni `if` e `else`, come tutte le istruzioni viste precedentemente, possono essere

include in funzioni.

```
def stampa_ombrello(mal_tempo):
    if mal_tempo:
        print("usciamo con l'ombrello")
    else:
        print("usciamo senza l'ombrello")

pioggia = False
nuvoloso = True
stampa_ombrello(pioggia or nuvoloso)
# Out: usciamo con l'ombrello

nuvoloso = False
stampa_ombrello(pioggia or nuvoloso)
# Out: usciamo senza l'ombrello
```

Spesso per prendere una decisione bisogna controllare più condizioni. Questo si potrebbe fare combinando `if` ed `else`, ma Python ha una forma estesa dell'`if` che è utile in tali situazioni.

```
if condizione_1:
    istruzioni_1
elif condizione_2:
    istruzioni_2
elif condizione_3:
    istruzioni_3
...
else:
    istruzioni_else
```

`elif` è l'istruzione che dichiara decisioni addizionali, che possono essere un numero arbitrario, tutte indentate allo stesso livello del primo `if`. Se `condizione_1` è `True`, si eseguono le `istruzioni_1` e poi si salta alla fine dell'intero blocco. Se invece `condizione_1` è `False` e `condizione_2` è `True`, si eseguono le `istruzioni_2` e poi si salta alla fine del blocco. Se `condizioni_1` e `condizione_2` sono `False` e `condizione_3` è `True`, si eseguono `istruzioni_3` e poi si salta alla fine del blocco. Se nessuna delle condizioni è soddisfatta, cioè nessuna è `True`, si eseguono le `istruzioni_else`.

```

def commenti_voto(voto):
    print("Il voto e'", voto)
    if voto < 18:
        print("mi dispiace")
    elif voto == 18:
        print("appena sufficiente")
    elif voto < 24:
        print("OK, ma potevi fare meglio")
    elif voto == 30:
        print("congratulazioni!")
    else:
        print("bene!")

commenti_voto(15)
# Out: Il voto e' 15
# Out: mi dispiace
commenti_voto(18)
# Out: Il voto e' 18
# Out: appena sufficiente
commenti_voto(23)
# Out: Il voto e' 23
# Out: OK, ma potevi fare meglio
commenti_voto(27)
# Out: Il voto e' 27
# Out: bene!
commenti_voto(30)
# Out: Il voto e' 30
# Out: congratulazioni!

```

Nei contesti in cui Python si aspetta una condizione booleana, ad esempio nelle istruzioni `if` ed `elif`, se le espressioni passate hanno tipi diversi, il linguaggio cercherà di interpretarle come booleani usando appropriate convenzioni. Ad esempio, lo zero per i numeri e la stringa vuota per le stringhe corrisponde a falso.

```

def stampa_vuoto(x):
    if x:
        print('valore non nullo')
    else:
        print('valore nullo')

```

```
stampa_vuoto(0)
# Out: valore nullo
stampa_vuoto(1)
# Out: valore non nullo
stampa_vuoto('')
# Out: valore nullo
stampa_vuoto('ciao')
# Out: valore non nullo
```

## 5.5 MODULI E PROGRAMMI

Nel codice scritto fino ad ora abbiamo incluso sia funzioni, che potremmo essere utili successivamente, che codice per testarle. Questi test sono convenienti, ma non vogliamo eseguirli ogni volta che importiamo il modulo. Ad esempio, salviamo l'ultimo blocco di codice in un modulo `programma.py`. Quando importiamo il modulo otterremo la stampa indesiderata dei valori dei test.

```
import programma
# Out: valore nullo
# Out: valore non nullo
# Out: valore nullo
# Out: valore non nullo
```

Per eseguire i test quando il programma viene eseguito da terminale direttamente, e non durante l'import, basta mettere l'esecuzione dei test in un blocco delimitato da `if __name__ == '__main__':`. Questo costrutto è idiomatico in Python e fa sì che la parte di codice dentro l'`if` sia eseguita solamente quando il modulo è eseguito direttamente. Questo funziona perché la variabile speciale `__name__` ha valore `'__main__'` quando il modulo è eseguito direttamente, mentre quando è importato ha valore uguale al nome del modulo stesso. Grazie a ciò possiamo scrivere il codice di testing all'interno del modulo stesso e possiamo comunque importare il modulo senza che il codice di testing venga eseguito. Nel nostro caso modifichiamo il codice precedente mettendo i test nel blocco delimitato.

```
if __name__ == '__main__':
    stampa_vuoto(0)
    stampa_vuoto(1)
    stampa_vuoto('')
    stampa_vuoto('ciao')
```

Se ora importiamo il modulo, salvato come `modulo.py`, non otteniamo la

stampa dei tests, ma possiamo usare le funzioni definite.

```
import modulo

stampa_vuoto(0)
# Out: valore nullo
```

## 6 SEQUENZE DI DATI

Negli esempi fatti fino ad ora, le variabili in Python hanno rappresentato valori singoli. È spesso necessario memorizzare sequenze di valori e lavorare su queste. Ad esempio, potremmo volere calcolare la temperatura media di una località a partire dalle temperature giornaliere. In questo capitolo introdurremo strumenti di programmazione per modellare sequenze di dati.

## 6.1 LISTE

Per memorizzare un insieme di valori potremmo usare una variabile per ogni valore. Ma questo metodo non è praticabile quando il numero di valori diventa alto. In tutti i linguaggi di programmazione esistono tipi di dati che permettono di raggruppare insieme un numero variabile di valori. Python ha svariati tipi per rappresentare sequenze ordinate di valori. Il modo più flessibile è quello di usare *liste* di tipo `list`, le quali possono essere scritte come un elenco di valori separati da virgole e racchiuso tra parentesi quadrate, e possono contenere valori di qualunque tipo. La lista vuota si indica non includendo nulla tra le parentesi.

```
primi = [2, 3, 5, 7, 11, 13, 17, 19]
print(primi)
# Out: [2, 3, 5, 7, 11, 13, 17, 19]
colori = ['blu', 'rosso', 'verde', 'giallo']
print(colori)
# Out: ['blu', 'rosso', 'verde', 'giallo']
misc = ['red', 2, 3.14, 'blue']
print(misc)
# Out: ['red', 2, 3.14, 'blue']

lista_vuota = []
print(lista_vuota)
# Out: []

print(type(colori))
# Out: <class 'list'>
```

## 6.2 TUPLE

A volte è utile rappresentare una sequenza di valori di lunghezza fissa. Per questo si usano le *tuple* di tipo `tuple`. Le tuple si indicano con una lista di valori racchiusa tra parentesi rotonde. In alcuni casi le parentesi possono essere omesse, anche se per i principianti è preferibile l'uso esplicito delle parentesi. Le tuple vuote sono indicate con parentesi tonde vuote.

```
tupla = ('ciao', 1, 2)
print(tupla)
# Out: ('ciao', 1, 2)

tupla_vuota = ()
print(tupla_vuota)
# Out: ()

print(type( (1,2) ))
# Out: <class 'tuple'>
```

Le tuple possono essere utilizzate per definire funzioni che ritornano più valori. La funzione `hms()`, definita in un capitolo precedente, faceva proprio questo. Un altro esempio è una funzione che calcola il quoziente e il resto della divisione.

```
def quoziente_resto(x,y):
    return x // y, x % y

q = quoziente_resto(5,2)
print(q)
# Out: (2, 1)
print(type(q))
# Out: <class 'tuple'>
```

## 6.3 OPERAZIONI SU SEQUENZE

Liste e tuple rappresentano sequenze di valori su cui si può operare con operatori e funzioni simili. Molte di queste operazioni hanno lo stesso significato sulle stringhe, che si possono considerare come sequenze di caratteri. L'operatore `+` concatena due sequenze, l'operatore `*` ripete una sequenza più volte e la funzione `len()` calcola la lunghezza di una sequenza.

```
saluto = 'Buon ' + 'giorno'
print(saluto)
# Out: Buon giorno
boom = 'tic tac '*5 + 'BOOM!'
print(boom)
# Out: tic tac tic tac tic tac tic tac tic tac BOOM!
len(boom)
# Out: 45

primi = [2, 3, 5, 7, 11, 13, 17, 19]
primi2 = primi + [23, 29]
print(primi2)
# Out: [2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
[1, 2, 3]*4
# Out: [1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
len(primi)
# Out: 8

persone = ('Bruno', 'Luisa') + ('Anna', 'Mario')
print(persone)
# Out: ('Bruno', 'Luisa', 'Anna', 'Mario')
len(persone)
# Out: 4
```

La libreria standard di Python include varie funzioni predefinite per la manipolazione di sequenze. Ad esempio, la funzione `sum()` somma i valori di una sequenza, mentre `all()` e `any()` ritornano `True` se rispettivamente tutti i valori, o almeno un valore, sono veri.

```
sum([2,3,4])
# Out: 9
all([False,True])
# Out: False
any([False,True])
# Out: True
```

Tra le funzioni predefinite più usate ci sono quelle di ordinamento, che permettono di calcolare il valore minimo `min()` e massimo `max()` di una sequenza, o di ordinare una sequenza di valori `sorted()`. L'ordinamento di numeri è in ordine decrescente, mentre l'ordinamento di stringhe segue l'ordine lessicografico, ma distingue tra maiuscole e minuscole, le prime della quali vengono sempre prima delle seconde.

```
lst = [10,1,20]
print( min(lst) )
# Out: 1
print( max(lst) )
# Out: 20
print( sorted(lst) )
# Out: [1, 10, 20]

rubrica = ['Bruno', 'Sofia', 'Anna', 'Mario']
print( sorted(rubrica) )
# Out: ['Anna', 'Bruno', 'Mario', 'Sofia']
```

Si possono convertire sequenze tra loro usando `list()` e `tuple()` per convertire sequenze in liste e tuple rispettivamente.

```
# conversione in liste e tuple
print(tuple([1,2]))
# Out: (1, 2)
print(list(('Bruno','Luisa')))
# Out: ['Bruno', 'Luisa']
print(list('Anna'))
# Out: ['A', 'n', 'n', 'a']
```

## 6.4 OPERATORI RELAZIONALI E DI APPARTENENZA

Gli operatori relazionali sono applicabili a tutti i tipi sequenza, quindi anche a liste e tuple. Il confronto è effettuato come per le stringhe considerando i valori delle sequenze da sinistra verso destra.

```
lst = [1, 2, 'abc', 5]
print(lst == [1, 2, 'abc', 5])
# Out: True
print(lst < [1, 2, 'abcd'])
# Out: True
```

L'operatore relazionale più usato per liste e tuple è l'operatore `in`, e il suo inverso `not in`, che permettono di testare l'appartenenza di un valore a una sequenza. Nel caso di tuple e liste l'appartenenza è di elementi singoli, mentre per le stringhe l'appartenenza è, come abbiamo visto, di sottostringhe.

```
seq = [1, 2, 3, 5, 8]
print(5 in seq)
# Out: True
print(4 in seq)
# Out: False
print('mela' in ['noce', 'mela', 'banana'])
# Out: True
print(4 not in seq)
# Out: True
```

Come per le stringhe, la lista e la tupla vuota sono interpretate come falso se usate in una espressione condizionale.

```
def stampa_vuoto(x):
    if x:
        print('valore non nullo')
    else:
```

```
print('valore nullo')

stampa_vuoto([])
# Out: valore nullo
stampa_vuoto([2,3,5])
# Out: valore non nullo
stampa_vuoto((2,3))
# Out: valore non nullo
```

L'esempio seguente mostra l'utilità dell'operatore d'appartenenza che può sostituire un lungo blocco di `if-elif`.

```
def frutto(x):
    if x == 'mela':
        return True
    elif x == 'pera':
        return True
    elif x == 'uva':
        return True
    else:
        return False

x = 'melo'
y = 'uva'
print(frutto('melo'))
# Out: False
print(frutto('uva'))
# Out: True

def frutto2(x):
    return x in ['mela', 'pera', 'uva']

print(frutto2('melo'))
# Out: False
print(frutto2('uva'))
# Out: True
```

Possiamo usare condizioni per validare i valore di ingresso di una funzione. Ad esempio, scriviamo una funzione che prende in input le prime tre lettere del nome di un mese e il giorno e ritorna `True` se la data è corretta rispetto

ad un anno non bisestile.

```
def check_date(mese, giorno):
    if giorno < 1: return False
    if mese == 'feb':
        return giorno <= 28
    elif mese in ['apr', 'giu', 'set', 'nov']:
        return giorno <= 30
    elif mese in ['gen', 'mar', 'mag', 'lug',
                  'ago', 'ott', 'dic']:
        return giorno <= 31
    else:
        return False

print(check_date('gen', 31))
# Out: True
print(check_date('feb', 29))
# Out: False
print(check_date('dic', 0))
# Out: False
print(check_date('mir', 1))
# Out: False
```

## 6.5 ELEMENTI DI SEQUENZE

Ogni elemento in una sequenza è univocamente specificato dal suo *indice*, cioè il suo numero d'ordine a partire da zero nella lista. Per accedere ad un elemento si pone l'indice dell'elemento tra parentesi quadre dopo un riferimento alla sequenza. Se si tenta di accedere ad un elemento non presente nella sequenza, si otterrà un errore.

```
colori = ['blu', 'rosso', 'verde', 'giallo']
print(colori[0])
# Out: blu
persone = ('Bruno', 'Sofia', 'Mario')
print(persone[1])
# Out: Sofia
stringa = 'abracadabra'
print(stringa[2])
# Out: r

print(colori[10])
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: IndexError: list index out of range
```

Le parentesi quadre accettano anche indici negativi, che in questo caso sono contati dalla fine verso l'inizio della sequenza, con `-1` che rappresenta l'ultima posizione.

```
colori = ['blu', 'rosso', 'verde', 'giallo']

# L'elemento in ultima posizione
print(colori[-1])
# Out: giallo
# Quello in penultima posizione
print(colori[-2])
# Out: verde
```

Gli elementi di una lista possono essere modificati assegnando un valore all'indice desiderato. Gli elementi di tuple e stringhe non possono essere mutati. Si dice che le liste sono *mutabili* mentre stringhe e tuple sono *immutabili*. Questa distinzione è fondamentale in Python ed è la vera ragione per cui esistono sia tuple che liste.

```
colori = ['blu', 'rosso', 'verde', 'giallo']
colori[1] = 'nero'
print(colori)
# Out: ['blu', 'nero', 'verde', 'giallo']
```

```
persone[1] = 'Sara'
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: TypeError: 'tuple' object does not support item assignment
stringa[5] = 'z'
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: TypeError: 'str' object does not support item assignment
```

## 6.6 UNPACKING DI VALORI

Possiamo accedere a più elementi di una sequenza con la notazione della parentesi quadre. Questo però rende in codice spesso poco leggibile e pronò ad essere perché si manipolano gli indici esplicitamente. Python ha una notazione piuttosto conveniente chiamata *sequence unpacking* che assegna simultaneamente tutti i valori di una sequenza ad altrettante variabili. Nelle istruzioni di assegnamento basta indicare più variabili separate da virgole. L'unpacking è maggiormente usato con le tuple, dato che queste mantengono il numero dei valori, ma può essere usato con un qualsiasi tipo sequenza.

```
persone = ('Bruno', 'Sofia', 'Mario')
print(persone)
# Out: ('Bruno', 'Sofia', 'Mario')

# sequence unpacking
p1, p2, p3 = persone
print(p1, p2, p3)
# Out: Bruno Sofia Mario

# non si accede con numero di variabili diverso
p1, p2 = persone
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: ValueError: too many values to unpack (expected 2)
```

L'unpacking, combinato con l'uso delle tuple, può essere usato per esprimere in modo succinto l'assegnamento di valori multipli, lo scambio dei valori di due variabili, e l'accesso ai valori multipli ritornati da funzioni.

```
# tupla senza parentesi tonde, seguito da unpacking
a, b = 'primo', 'secondo'
print(a, b)
# Out: primo secondo
```

```
# scambio di valori
b, a = a, b
print(a, b)
# Out: secondo primo

def quoziente_resto(a, b):
    return a/b, a%b
q, r = quoziente_resto(5, 3)
print(q, r)
# Out: 1.6666666666666667 2
```

## 6.7 SOTTOSEQUENZE O SLICES

In Python la sintassi delle parentesi quadre permette di ottenere una qualsiasi sottosequenza di elementi consecutivi specificando l'indice d'inizio e quello di fine. La sottosequenza `[a:b]` indica gli elementi da `a` e `b-1`. Nel gergo di Python questa sintassi si chiama *slice*. Gli indici di inizio e fine sono opzionali, e se mancanti indicano implicitamente il primo e l'ultimo elemento della sequenza. Gli indici possono essere negativi, come in precedenza.

```
colori = ['blu', 'rosso', 'verde', 'giallo']

# sottolista dalla posizione 1 alla 2
print(colori[1:3])
# Out: ['rosso', 'verde']
# sottolista dall'inizio alla penultima
print(colori[0:-1])
# Out: ['blu', 'rosso', 'verde']
# sottolista dalla posizione 2 in poi
print(colori[2:])
# Out: ['verde', 'giallo']
# sottolista dall'inizio alla pos. 2
print(colori[:2])
# Out: ['blu', 'rosso']
# copia dell'intera lista
print(colori[:])
# Out: ['blu', 'rosso', 'verde', 'giallo']
```

La sintassi delle sottosequenze si applica a tutte le sequenze viste, incluse tuple e stringhe.

```
s = 'Questa è proprio una bella giornata.'
# sottostringa dalla posizione 0 alla 5
print(s[0:6])
# Out: Questa
# sottostringa dalla posizione 7 in poi
print(s[7:])
```

```
# Out: è proprio una bella giornata.  
# sottostringa dall'inizio fino alla posizione 15  
print(s[:16])  
# Out: Questa è proprio
```

Per le liste, le slice possono essere usate anche per l'assegnamento, dove la sottolista specificata dalla slice è sostituita con la lista assegnatagli. Possiamo inserire o eliminare valori assegnando liste con lunghezza diverse dalla sottolista selezionata.

```
colori = ['blu', 'rosso', 'verde', 'giallo']  
  
# cambiamento di valori  
colori[1:3] = ['arancio', 'viola']  
print(colori)  
# Out: ['blu', 'arancio', 'viola', 'giallo']  
  
# inserimento di valori  
colori[1:1] = ['nero']  
print(colori)  
# Out: ['blu', 'nero', 'arancio', 'viola', 'giallo']  
  
# cancellazione di valori  
colori[1:2] = []  
print(colori)  
# Out: ['blu', 'arancio', 'viola', 'giallo']
```

Per finire facciamo un semplice esempio che usa le slice per suddividere un codice fiscale nelle sue componenti.

```
def print_cf(cf):  
    print('Cognome (tre lettere):', cf[0:3])  
    print('Nome (tre lettere):', cf[3:6])  
    print('Data di nascita e sesso:', cf[6:11])  
    print('Comune di nascita:', cf[11:15])  
    print('Carattere di controllo:', cf[-1])  
  
# codice fiscale fasullo  
print_cf('COGNOMDATASCOMUX')  
# Out: Cognome (tre lettere): COG
```

*# Out: Nome (tre lettere): NOM*  
*# Out: Data di nascita e sesso: DATAS*  
*# Out: Comune di nascita: COMU*  
*# Out: Carattere di controllo: X*

## 7 ITERAZIONE SU SEQUENZE

La necessità più comune quando si manipolano sequenze è di applicare una stessa sequenza di istruzioni per ogni elemento della sequenza, come ad esempio, stampare tutti i numeri in una lista. In questo capitolo introdurremo varie istruzioni che permetteranno iterare su sequenze di dati.

## 7.1 ITERAZIONE SU SEQUENZE

L'istruzione `for` permette di *iterare* sugli elementi di una sequenza ripetendo un blocco di istruzioni per ogni elemento. La sintassi generale è

```
for var in seq:
    istruzioni
```

dove `for` e `in` sono parole chiavi. Il ciclo `for` scorre i valori della sequenza `seq` in ordine, e ad ogni iterazione assegna alla variabile `var` il prossimo valore così che le `istruzioni`, indentate di 4 spazi, all'interno del `for` possano elaborare il valore. Ad esempio, possiamo stampare i valori di una lista utilizzando la funzione `print()` all'interno di un `for`.

```
colori = ['blu', 'rosso', 'verde', 'giallo']
for colore in colori:
    print(colore)
# Out: blu
# Out: rosso
# Out: verde
# Out: giallo
```

Facciamo ora alcuni esempi di iterazione. Scriviamo una funzione che calcola la somma degli elementi di una lista. Per questa funzione utilizzeremo una variabile aggiuntiva, chiamata *contatore*, che mantiene la somma parziale. La nostra funzione è equivalente alla funzione predefinita `sum()`.

```
def somma(valori):
    '''Ritorna la somma dei valori.'''
    s = 0
    for valore in valori:
        s += valore
    return s

print(somma([1,2,3]))
```

```
# Out: 6
```

Per capire cosa sta succedendo nel ciclo è utile stampare dei valori mentre la funzione esegue.

```
def somma_stampa(valori):
    s = 0
    for valore in valori:
        print('aggiungi',valore,'a',s)
        s += valore
        print('valore aggiunto',s)
    return s

print(somma_stampa([1,2,3]))
# Out: aggiungi 1 a 0
# Out: valore aggiunto 1
# Out: aggiungi 2 a 1
# Out: valore aggiunto 3
# Out: aggiungi 3 a 3
# Out: valore aggiunto 6
# Out: 6
```

Scriviamo ora una funzione che emula il comportamento della funzione `len()`.

```
def lunghezza(valori):
    '''Ritorna la lunghezza di una sequenza.'''
    l = 0
    for valore in valori:
        l += 1
    return l

print(lunghezza([1,2,3]))
# Out: 3
```

Scriviamo una funzione che prende in input una lista con valori numerici, e ritorna una nuova lista i cui valori sono quelli della lista di input elevati al cubo. Per farlo useremo una lista aggiuntiva che a cui aggiungiamo valori mentre scorriamo la lista iniziale.

```

def cubi(valori):
    '''Ritorna una nuova lista che contiene i cubi
    dei valori della lista di input.'''
    cubi = []
    for valore in valori:
        cubi += [valore**3]
    return cubi

primi = [2, 3, 5, 7, 11, 13]
cubi_primi = cubi(primi)
print(primi)
# Out: [2, 3, 5, 7, 11, 13]
print(cubi_primi)
# Out: [8, 27, 125, 343, 1331, 2197]

```

Per capire cosa sta succedendo nel ciclo è utile stampare dei valori mentre la funzione esegue.

```

def cubi_stampa(valori):
    '''Ritorna una nuova lista che contiene i cubi
    dei valori della lista di input.'''
    cubi = []
    for valore in valori:
        cubo = valore**3
        print('aggiungi',cubo,'a',cubi)
        cubi += [cubo]
        print('valore aggiunto',cubi)
    return cubi

primi = [2, 3, 5]
cubi_primi = cubi_stampa(primi)
# Out: aggiungi 8 a []
# Out: valore aggiunto [8]
# Out: aggiungi 27 a [8]
# Out: valore aggiunto [8, 27]
# Out: aggiungi 125 a [8, 27]
# Out: valore aggiunto [8, 27, 125]

```

Scriviamo infine una funzione che emula l'operatore di concatenamento `+`.

```
def concatena(valori1, valori2):  
    valori = []  
    for valore in valori1:  
        valori += [valore]  
    for valore in valori2:  
        valori += [valore]  
    return valori  
  
print(concatena([1,2,3],[4,5,6]))  
# Out: [1, 2, 3, 4, 5, 6]
```

## 7.2 ITERAZIONE SU SEQUENZE DI INTERI

La funzione predefinita `range()` ci permette di iterare su una sequenza di numeri interi. La funzione `range()` accetta un numero variabile di parametri. Quando è chiamata con un solo parametro, `range(n)` itera sugli interi a `0` a `n-1`. Con due parametri, `range(a,b)` itera da `a` a `b-1`. Il terzo parametro permette di incrementare le sequenza di valori superiori a 1.

```
for i in range(3):
    print(i)
# Out: 0
# Out: 1
# Out: 2

for i in range(2,5):
    print(i)
# Out: 2
# Out: 3
# Out: 4

for i in range(2,5,2):
    print(i)
# Out: 2
# Out: 4
```

La funzione `range()` definisce una sequenza speciale che può essere convertita in altre sequenze, applicando le operazioni `list()` o `tuple()`.

```
list(range(2,20,3))
# Out: [2, 5, 8, 11, 14, 17]
```

Possiamo scrivere una funzione che prende in input un intero `n` e crea una lista che contiene i cubi degli interi da `1` a `n`.

```
def crea_cubi(n):
```

```

"""Ritorna una lista che contiene i cubi degli
interi da 1 a n."""
lst = []
for i in range(1, n + 1):
    lst = lst + [i**3]
return lst

lst = crea_cubi(10)
print(lst)
# Out: [1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]

```

Un utilizzo comune della funzione `range()` è quello di iterare sugli indici di una lista per poterne modificare i valori, anche se in generale, in Python è preferibile creare nuove liste e non modificare quelle già esistenti.

```

primi = [2, 3, 5, 7, 11, 13]
for indice in range(len(primi)):
    primi[indice] = primi[indice] ** 2

print(primi)
# Out: [4, 9, 25, 49, 121, 169]

```

Adesso possiamo scrivere una variazione della funzione `setta_cubi()` in cui modifichiamo direttamente la lista di input.

```

def setta_cubi(lst):
    """Modifica la lista di input elevando al
    cubo tutti i suoi valori."""
    for i in range(len(lst)):
        lst[i] = lst[i]**3

primi = [2, 3, 5, 7, 11, 13]
print(primi)
# Out: [2, 3, 5, 7, 11, 13]

setta_cubi(primi)
print(primi)
# Out: [8, 27, 125, 343, 1331, 2197]

```

Questo comportamento può apparire strano, ma segue dal modo in cui

Python definisce il passaggio dei parametri a funzioni. Il parametro `lst` della funzione `setta_cubi()` non riceve una copia della variabile `primi`, ma il suo valore stesso. Quindi lo può modificare e la modifica è visibile all'esterno della funzione. Questo comportamento è corretto in Python ma vivamente sconsigliato perché fonte di innumerevoli errori di programmazioni. Se si vuole alterare una lista, è spesso meglio crearne una nuova, ad esempio con la funzione `cubi()`, e poi assegnare alla variabile la nuova lista.

## 7.3 ITERAZIONE CON CONDIZIONALI

Usando l'istruzione `if` possiamo scrivere una funzione che emula l'operatore `in`. In questo caso useremo l'istruzione `return` dentro ad un ciclo eseguita solo quando la condizione è vera.

```
def appartiene(valore_test, valori):
    '''Ritorna True se valore_test è in valori,
    False altrimenti.'''
    for valore in valori:
        if valore == valore_test:
            # usciamo dal ciclo e dalla funzione
            # inserendo return nel ciclo
            return True
    return False

lst = ['mela', 'pera', 'uva']
print('arancia' in lst, appartiene('arancia',lst))
# Out: False False
print('mela' in lst, appartiene('mela',lst))
# Out: True True
```

Scriviamo una funzione che emula la funzione predefinita `max()`. Per farlo manterremo in una variabile aggiuntiva il valore massimo trovato durante lo scorrimento della lista.

```
def valore_massimo(valori):
    '''Ritorna il valore massimo di una lista.'''
    if not valori: return None # lista vuota
    massimo = valori[0]
    for valore in valori:
        if valore > massimo:
            massimo = valore
    return massimo

print(valore_massimo([10,1,20]))
# Out: 20
```

---

Scriviamo una funzione che ritorna l'indice di un elemento di una lista, se esiste, o `-1` se non esiste.

```
def indice(valori, elemento):
    '''Ritorna l'indice di un elemento,
    o -1 se non esiste.'''
    for i in range(len(valori)):
        if valori[i] == elemento:
            return i
    return -1

print(indice([10,1,20],20))
# Out: 2
print(indice([10,1,20],30))
# Out: -1
```

Scriviamo una funzione che verifica se una lista è ordinata. Per farlo, basta confrontare tutte le coppie di elementi consecutivi utilizzando `range()` per iterare sul primo indice.

```
def ordinata(lst):
    '''Ritorna True se la list lst è ordinata,
    altrimenti False.'''
    for i in range(len(lst)-1):
        if lst[i] > lst[i+1]:
            return False
    return True

lst = [2,3,1]
print(ordinata(lst))
# Out: False
print(ordinata(sorted(lst)))
# Out: True
```

Con una piccola modifica, scriviamo una funzione che ritorna l'indice del primo valore di una lista che non è compatibile con l'ordinamento della stessa.

```
def elemento_non_ordinato(lst):  
    '''Ritorna l'indice del primo valore che non  
    rispetta l'ordinamento.'''  
    for i in range(len(lst)):  
        if lst[i] > lst[i+1]:  
            return i  
    return -1  
  
print(elemento_non_ordinato([1,2,3,2]))  
# Out: 2  
print(elemento_non_ordinato(['abc', 'abcc', 'ab']))  
# Out: 1
```

## 7.4 CONTROLLO DELL'ITERAZIONE

L'istruzione `if` permette di alterare l'esecuzione di cicli come abbiamo visto nei vari esempi precedenti dove le condizioni venivano usate per escludere o includere singole iterazioni in un ciclo. L'utilizzo di `for` e `if` insieme è così comune che Python prevede due istruzioni aggiuntive, `continue` e `break`, per supportare cicli con condizionali. L'istruzione `continue` salta le rimanenti istruzioni del blocco e continua alla prossima iterazione. L'istruzione `break` termina il ciclo senza considerare gli elementi successivi.

Possiamo scrivere ad esempio una funzione che prese in input due liste e ritorna la media degli elementi della prima lista che non appartengono alla seconda.

```
def media_esclusiva(lst, nolst):
    '''Media degli elementi in lst e non in nolst'''
    total = 0
    count = 0
    for s in lst:
        if s in nolst:
            # Salta alla prossima iterazione
            continue
        total += s
        count += 1
    return total / count

print(media_esclusiva([2,3,4],[3]))
# Out: 3.0
```

Come ultimo esempio scriviamo una funzione che prende in input una lista e ritorna la media degli elementi calcolata fino ad incontrare il primo numero negativo.

```
def media_esclusiva2(lst):
    total = 0
    count = 0
```

```
for s in lst:
    if s < 0:
        # Esci dal ciclo
        break
    total += s
    count += 1
return total / count

print(media_esclusiva2([2,3,4,-1,10,20]))
# Out: 3.0
```

## 7.5 ITERAZIONE SU CONDIZIONE

Il `for` itera su elementi di una sequenza. Il `while` permette invece di iterare finché una condizione non risulta falsa. La sintassi generale del `while` è

```
while cond:
    istruzioni
```

dove il blocco `istruzioni` sarà eseguito finché la condizione `cond` risulta vera. Se non si vuole avere un ciclo infinito occorrerà che l'esecuzione del blocco `istruzioni`, ad una certa iterazione, faccia sì che la condizione `cond` diventi falsa. Il `while` risulta utile quando bisogna iterare un blocco di istruzioni non sapendo, a priori, quale sarà il numero di iterazioni. Ad esempio, se vogliamo determinare se un numero è primo, cioè è divisibile solamente per 1 e per se stesso, possiamo usare un `while` per provare i possibili divisori.

```
def primo(n):
    '''Ritorna True se il numero è primo.'''
    k = 2
    while k < n and (n % k) != 0:
        k += 1
    return k == n

print(primo(10))
# Out: False
print(primo(13))
# Out: True
```

Ovviamente si poteva scrivere anche usando il `for` ma così è più chiaro che ci fermiamo non appena troviamo un divisore. Un esempio in cui la convenienza di usare un `while` è ancora più evidente è trovare il più piccolo numero primo maggiore o uguale ad un intero dato.

```
def primo_minimo(m):
```

```
'''Ritorna il più piccolo primo >= m.'''  
while not primo(m):  
    m += 1  
return m  
  
print(primo_minimo(10))  
# Out: 11
```

## 7.6 ITERAZIONE E UNPACKING

Abbiamo visto come possiamo accedere agli elementi delle sequenze con l'unpacking che assegna simultaneamente tutti i valori della sequenza ad altrettante variabili con nomi diversi separate da virgole. Possiamo combinare iterazione e unpacking insieme per iterare direttamente su sequenze di tuple.

```
numeri = [(1, 'uno'), (2, 'due')]

# numero è una tupla
for numero in numeri:
    print(numero)
# Out: (1, 'uno')
# Out: (2, 'due')

# unpacking delle tuple esplicito
for numero in numeri:
    valore, stringa = numero
    print(valore, stringa)
# Out: 1 uno
# Out: 2 due

# unpacking delle tuple implicito
for valore, stringa in numeri:
    print(valore, stringa)
# Out: 1 uno
# Out: 2 due
```

Come abbiamo visto finora, c'è spesso la necessità di iterare su elementi di una sequenza e di conoscerne l'indice alla stesso tempo. La funzione predefinita `enumerate()` permette di iterare su coppie indice-valore.

```
colori = ['blu', 'rosso', 'verde', 'giallo']
for indice, colore in enumerate(colori):
    print(indice, colore)
```

```
# Out: 0 blu  
# Out: 1 rosso  
# Out: 2 verde  
# Out: 3 giallo
```

## 7.7 COMPREHENSIONS

L'iterazione è usata per trasformare una sequenza in un'altra, ad esempio in `cubi()`, o creare una sequenza con `range()`. Manipolazioni di questo tipo sono molto frequenti in Python. Per questo il linguaggio introduce una notazione succinta chiamata *list comprehension*. La sintassi generale è

```
[espressione for variable in lista if condizione]
```

che è equivalente alla funzione

```
def comprehension(lista):
    ret = []
    for variabile in lista:
        if condizione:
            ret += [ espressione ]
    return ret
```

Con questa notazione possiamo esprimere alcune variazioni delle funzioni precedenti.

```
# lista dei quadrati dei numeri tra 1 e 5
quadrati = [i ** 2 for i in range(5)]
print(quadrati)
# Out: [0, 1, 4, 9, 16]

# sottolista dei numeri pari
pari = [i for i in quadrati if i % 2 == 0]
print(pari)
# Out: [0, 4, 16]
```

## 8 OGGETTI E METODI

Finora abbiamo usato valori in Python senza preoccuparci di come solo rappresentati internamente dal linguaggio. In questo capitolo introdurremo il concetto di oggetti, che in Python sono usati per rappresentare tutti i valori.

## 8.1 OGGETTI, TIPI E IDENTITÀ

In Python è un linguaggio *orientato agli oggetti*, dove ogni valore è un *oggetto*. Un oggetto ha un *valore*, un *tipo* e una *identità*. Il valore dell'oggetto `'stringa'` è la sequenza di caratteri "stringa", il tipo è `str` e l'identità è determinata dalla locazione in memoria dell'oggetto. Similmente l'oggetto `7` ha come valore il numero sette, come tipo `int` e una sua identità propria. Ovviamente due oggetti differenti non possono occupare la stessa locazione in memoria, quindi due oggetti con la stessa identità sono esattamente lo stesso oggetto. Però oggetti differenti, cioè con differenti identità, possono avere lo stesso tipo e valore. Abbiamo già visto come `type()` ritorna il tipo di un oggetto.

```
# tipi base
print(type('stringa'))
# Out: <class 'str'>
print(type(13))
# Out: <class 'int'>
print(type(12345678901234))
# Out: <class 'int'>
print(type(13.0))
# Out: <class 'float'>
print(type([1,2,3]))
# Out: <class 'list'>
```

La funzione predefinita `id(obj)` ritorna l'identità dell'oggetto `obj`, espressa da un intero che generalmente è l'indirizzo in memoria di `obj`.

```
x = [1, 2]
y = [1, 2]

print(id(x))
# Out: 4315311432
print(id(y))
# Out: 4315351752
```

```
# gli oggetti x e y hanno lo stesso valore
# ma hanno identità differenti

print(x == y)
# Out: True
print(id(x) == id(y))
# Out: False
```

L'operatore `==` controlla l'uguaglianza di valore non l'uguaglianza d'identità. Può capitare che oggetti immutabili con lo stesso valore, ma creati in posti diversi, abbiano la stessa identità. Per ragioni di efficienza Python può usare lo stesso oggetto immutabile invece che crearne uno nuovo. Questo non produrrà nessun problema proprio perché sono oggetti il cui valore non può cambiare mai.

```
x = 'stringa'
y = 'stringa'
print(id(x))
# Out: 4315384832
print(id(y))
# Out: 4315384832
# x e y sono lo stesso oggetto
```

Un tipo, ciò che è ritornato dalla funzione `type()`, è anch'esso un oggetto ma non approfondiremo oltre questo aspetto. Come abbiamo visto in precedenza, i nomi dei tipi possono essere usati come *costruttori* per creare oggetti di quel tipo e per testare il tipo di un oggetto.

```
print(int('00123'))
# Out: 123
print(int(12.3))
# Out: 12
print(str(12.3))
# Out: 12.3
print(list())
# Out: []
```

C'è un oggetto speciale `None` che è usato per indicare l'assenza di un valore ed è anche ritornato da un funzione quando l'esecuzione non termina

tramite un `return` o il `return` non ha espressione.

```
x = None
print(x)
# Out: None
print(type(x))
# Out: <class 'NoneType'>

# la funzione non ritorna nulla
def printme(val):
    val + 1

print(printme(1))
# Out: None
```

## 8.2 OGGETTI FUNZIONE

In Python, le funzioni stesse sono oggetti di `function` che hanno come valore una codifica delle istruzioni e hanno una operazione definita che è la chiamata.

```
def cubo(x): return x**3
def quadrato(x): return x**2

print(cubo)
# Out: <function cubo at 0x101364ea0>
print(id(cubo))
# Out: 4315303584
print(type(cubo))
# Out: <class 'function'>
cubo == quadrato
# Out: False
```

Questo permette di passare funzioni ad altre funzioni e memorizzare funzioni in variabili in modo completamente naturale. Vedremo alcuni esempi molto semplici.

```
# memorizza una funzione in una variabile
f = cubo
# chiama la funzione usando il nome della variabile
f(3)
# Out: 27

# funzioni come argomento
def print_f(f,x):
    print(f(x))

# passaggio di funzioni come argomenti
print_f(cubo,5)
# Out: 125
print_f(quadrato,5)
# Out: 25
```

---

La nozione di poter trattare funzioni come valori è molto espressiva, e è uno dei cardini di uno stile di programmazione detto *programmazione funzionale*. Questo libro non tratta in modo esplicito di questo tipo di programmazione, ma utilizzeremo la possibilità di passare funzioni come argomenti quando necessario per risolvere problemi specifici.

## 8.3 ASSEGNAMENTO DI OGGETTI

Torniamo a rivedere il concetto di assegnamento alla luce dell'introduzione degli oggetti. Come abbiamo visto possiamo creare due oggetti diversi con lo stesso valore. Quando assegnamo un oggetto ad un'altra variabile, non creiamo una copia dell'oggetto, ma assegnamo l'oggetto stesso. Per questo si dice che assegnamo un *riferimento* all'oggetto. Questo si può verificare guardando l'identità degli oggetti.

```
l1 = [1,2]
l2 = [1,2]
l3 = l1

# l1 e l3 si riferiscono allo stesso oggetto
print(id(l1),id(l2),id(l3))
# Out: 4315398280 4315351752 4315398280

# le modifiche di l1 saranno visibili on l3
print(l1,l2,l3)
# Out: [1, 2] [1, 2] [1, 2]
l1[0] = -1
print(l1,l2,l3)
# Out: [-1, 2] [1, 2] [-1, 2]
```

Questo comportamento può causare innumerevoli errori per i tipi mutabili perché le modifiche di un oggetto si ripercuotono su tutte le variabili che si riferiscono a quell'oggetto. Questa è la ragione per cui è consigliabile creare nuovi oggetti, ad esempio nel corpo delle funzioni, invece di alterare quelli già esistenti. Per i tipi immutabili questo non è problematico dato che i valori non possono cambiare una volta creati.

## 8.4 METODI

Intuitivamente, si può dire che un tipo permette di raccogliere tutti gli oggetti che appartengono ad una stessa “famiglia”. Gli oggetti dello stesso tipo possono avere valori differenti ma condividono le operazioni che possiamo effettuare su di essi. Tutti gli interi possono essere sommati, moltiplicati, elevati a potenza, ma non hanno una lunghezza. Le stringhe hanno una lunghezza, possono essere concatenate, ma non possono essere elevate a potenza.

Python offre una sintassi speciale per funzioni specifiche ad un determinato tipo. Queste funzioni si chiamano *metodi* e vengono chiamati, o “invocati”, con la sintassi `obj.metodo(parametri)` dove `obj` è l’oggetto su cui eseguire l’operazione, `metodo` il nome dell’operazione e `parametri` i suoi parametri. Se il metodo fosse una funzione verrebbe chiamata con `metodo(obj, parametri)`. Vedremo successivamente come definire nuovi tipi e i relativi metodi. Per adesso ci limiteremo a considerare i metodi più utili di alcuni tipi predefiniti. Si può ottenere l’elenco dei metodi supportati da un tipo chiamando la funzione `help(tipo)`. Un esempio di metodo è il `is_integer` definito per il tipo `float`, ma non definito ad esempio per il tipo `int`.

```
x = 1.5
print(x, x.is_integer())
# Out: 1.5 False
x = 1.0
print(x, x.is_integer())
# Out: 1.0 True
y = 13
print(y, y.is_integer())
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: AttributeError: 'int' object has no attribute 'is_integer'
```

## 8.5 METODI DELLE LISTE

Il metodo `count()` ritorna il numero di occorrenze di un valore nella lista, mentre il metodo `index()` ritorna la posizione della prima occorrenza di un valore nella lista.

```
lst = ['uno', 1, 'uno', 'due', 2, 1]
print(lst.count(1))
# Out: 2
print(lst.count('uno'))
# Out: 2

# ritorna l'indice della prima occorrenza di 'due'
print(lst.index('due'))
# Out: 3
# inizia la ricerca dalla posizione 2
print(lst.index(1, 2))
# Out: 5

# errore: non trova il valore 3
print(lst.index(3))
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: ValueError: 3 is not in list
```

Altri metodi permettono di modificare una lista aggiungendo, inserendo o rimuovendo valori nella lista stessa, senza cioè crearne un'altra. Il metodo `append()` aggiunge un valore in coda alla lista stessa. Il metodo `insert()` inserisce un valore in una certa posizione all'interno della lista. Il metodo `remove()` rimuove la prima occorrenza di un valore e se non c'è genera un errore, mentre il metodo `pop()` rimuove il valore in una posizione e lo ritorna, di default la posizione è l'ultima.

```
lst = ['uno', 1, 'uno', 'due', 2, 1]
lst.append('tre')
print(lst)
```

```

# Out: ['uno', 1, 'uno', 'due', 2, 1, 'tre']
lst1 = lst + ['quattro']
lst == lst1
# Out: False

# inserisce 'sei' in posizione 2, spostando gli
lst.insert(2, 'sei')
print(lst)
# Out: ['uno', 1, 'sei', 'uno', 'due', 2, 1, 'tre']
lst.insert(0, 'primo')
print(lst)
# Out: ['primo', 'uno', 1, 'sei', 'uno', 'due', 2, 1, 'tre']

lst.remove('uno')
print(lst)
# Out: ['primo', 1, 'sei', 'uno', 'due', 2, 1, 'tre']

x = lst.pop()
print(lst)
# Out: ['primo', 1, 'sei', 'uno', 'due', 2, 1]
x = lst.pop(2)
print(lst)
# Out: ['primo', 1, 'uno', 'due', 2, 1]

```

Come esempio vogliamo scrivere una funzione `rot()` che ruota una lista di una posizione verso destra. Ad esempio se la lista di input è `[1,2,3,4,5]` la funzione la modifica così `[5,1,2,3,4]`. Possiamo poi usare `rot()` per scrivere una funzione `rotate()` che prende in input anche un intero `k` e ruota la lista `k` volte.

```

def rot(lst):
    '''Ruota la lista data di una posizione
    a destra.'''
    last = lst.pop()
    lst.insert(0, last)

lst = [1,2,3,4,5]
rot([1,2,3,4,5])
print(lst)
# Out: [1, 2, 3, 4, 5]

```

```
def rotate(lst, k):
    '''Ruota la lista data di k posizioni
    a destra.'''
    for _ in range(k):
        rot(lst)

def test_rotate(lst, k):
    print(lst, " k =", k)
    rotate(lst, k)
    print(lst)

lst = [1,2,3,4,5]
rotate(lst, 3)
print(lst)
# Out: [3, 4, 5, 1, 2]
```

# 9

## TABELLE DI DATI

I Computer sono particolarmente utili per elaborare dati. Fino ad ora abbiamo visto come memorizzare ed elaborare sequenze di dati con liste. In questo capitolo introdurremo i dizionari che permettono di memorizzare tabelle di dati e di associare dati ad altri dati.

## 9.1 DIZIONARI

Un *dizionario* è un oggetto di tipo `dict` che può essere visto come una collezione di coppie *chiave-valore*. In un dizionario ad ogni chiave è univocamente associato un singolo valore. Quindi non possono esistere due chiavi uguali. Dalla chiave si può accedere direttamente al valore, ma non viceversa. Ad esempio, la chiave potrebbero essere il codice fiscale e il valore il nome e cognome.

Un dizionario si può creare mettendo, tra parentesi graffe, una lista di coppie chiave-valore separati dal simbolo `:`. Il dizionario vuoto viene creato con le parentesi graffe senza elementi. Nei dizionari, chiavi e valori possono avere tipi arbitrari. Ad esempio potremmo creare un dizionario che associa ad un nome un numero di telefono.

```
# dizionario con chiavi-valori
rubrica = { 'Sergio': '123456', 'Bruno': '654321' }
print(rubrica)
# Out: {'Sergio': '123456', 'Bruno': '654321'}
print(type(rubrica))
# Out: <class 'dict'>

# dizionari vuoti
print({})
# Out: {}
```

Possiamo accedere ai valori associati alle chiavi usando la sintassi delle parentesi quadre, similmente alle liste, ma specificando il valore della chiave. Se si accede ad un elemento non esistente si riceve un errore.

```
# valore associato a 'Sergio'
print(rubrica['Sergio'])
# Out: 123456

# valore associato a Bruno
```

```
print(rubrica['Bruno'])
# Out: 654321

# valore non esistente
print(rubrica['Giovanni'])
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error: KeyError: 'Giovanni'
```

Una volta che un dizionario è stato creato possiamo aggiungere, modificare o rimuovere associazioni. Per aggiungere una nuova chiave e il relativo valore si può sempre usare la sintassi delle parentesi quadre. Per rimuovere un chiave, basta usare il metodo `pop()`.

```
# aggiunge una nuova associazione
rubrica['Mario'] = '112233'
print(rubrica)
# Out: {'Sergio': '123456', 'Mario': '112233', 'Bruno': '654321'}

# modifica un'associazione esistente
rubrica['Sergio'] = '214365'
print(rubrica)
# Out: {'Sergio': '214365', 'Mario': '112233', 'Bruno': '654321'}

# rimuove un chiave e il corrispondente valore
x = rubrica.pop('Bruno')
print(rubrica)
# Out: {'Sergio': '214365', 'Mario': '112233'}
```

Si può controllare se una chiave è presente nel dizionario con l'operatore `in`. Si noti che questo è diverso dalle liste dove `in` controlla se un valore è presente. `len()` ritorna il numero di chiavi.

```
print('Sergio' in rubrica)
# Out: True
print('Luigi' in rubrica)
# Out: False
print('214365' in rubrica)
# Out: False
print(len(rubrica))
```

```
# Out: 2
```

Possiamo iterare sulle chiavi di un dizionario usando il dizionario direttamente in un ciclo `for` o utilizzando il metodo `keys()`. Possiamo iterare sulle coppie chiave-valore chiamando il metodo `items()`. Meno comunemente, si può iterare sui valori con `values()`, ma in questo caso non è possibile accedere alla chiave direttamente. Si noti che quando si itera sui dizionari le chiavi non sono ordinate.

```
# itera sulle chiavi di rubrica
for chiave in rubrica:
    print(chiave, rubrica[chiave])
# Out: Sergio 214365
# Out: Mario 112233

# itera sulle chiavi di rubrica
for chiave in rubrica.keys():
    print(chiave, rubrica[chiave])
# Out: Sergio 214365
# Out: Mario 112233

# itera sulle coppie chiavi-valori di rubrica
for chiave, valore in rubrica.items():
    print(chiave, valore)
# Out: Sergio 214365
# Out: Mario 112233

# itera sui valori di rubrica
for valore in rubrica.values():
    print(valore)
# Out: 214365
# Out: 112233
```

Per ottenere esplicitamente le liste di chiavi, valori o delle coppie chiave-valore, utilizziamo i metodi `keys()`, `values()` e `items()` passandoli alla funzione `list()`. Il risultato dei primi due casi saranno liste di elementi singoli, mentre il risultato di `list(items())` sarà una lista di tuple.

```
print(rubrica.keys())
```

```
# Out: dict_keys(['Sergio', 'Mario'])
print(rubrica.values())
# Out: dict_values(['214365', '112233'])
print(rubrica.items())
# Out: dict_items([('Sergio', '214365'), ('Mario', '112233')])
```

Come esempio scriviamo una funzione `aggiorna_dict(d, d2)` che aggiorna il dizionario `d` aggiungendo ad esso le associazioni di `d2`. Questa funzione è simile al metodo predefinito `update()`.

```
def aggiorna_dict(d, d2):
    '''Aggiunge a d gli elementi in d2.'''
    for k, v in d2.items():
        d[k] = v

rubrica = { 'Sergio': '112233',
            'Mario': '654321' }

rubrica2 = { 'Sergio': '333333',
             'Maria': '222222' }

aggiorna_dict(rubrica, rubrica2)
print(rubrica)
# Out: {'Sergio': '333333', 'Maria': '222222', 'Mario': '654321'}
```

Come per le liste, i dizionari si possono creare attraverso *comprehensions*. In questo caso la sintassi utilizza le parentesi graffe e le coppie chiave-valore separata da `:`.

```
cubi = { i: i**3 for i in range(3) }
print(cubi)
# Out: {0: 0, 1: 1, 2: 8}
```

## 9.2 INSIEMI

A volte è necessario memorizzare una serie di valori non ripetuti e non ordinati. In Python questo si fa con il tipo insieme o `set`. A differenza delle liste, un oggetto di tipo `set` non contiene duplicati e gli elementi sono in ordine arbitrario. Un insieme è quindi simile a un dizionario con le sole chiavi e senza valori. In generale, gli insiemi sono usati meno spesso dei dizionari, ma rimangono utili. Un insieme di valori si crea mettendo la lista dei valori tra parentesi graffe, mentre l'insieme vuoto si crea con `set()`.

```
# insieme con due valori
insieme = { 5, 'five' }
print(insieme)
# Out: {'five', 5}

# insiemi vuoti
print(set())
# Out: set()
```

Possiamo aggiungere elementi tramite il metodo `add()` e rimuoverli con i metodi `remove()` e `pop()`. L'aggiunta di un elemento già esistente non ha effetto.

```
insieme.add(6)
print(insieme)
# Out: {'five', 5, 6}

insieme.add(6)
print(insieme)
# Out: {'five', 5, 6}

insieme.remove('five')
print(insieme)
# Out: {5, 6}
```

Le istruzioni `for`, `in` e la funzione `len()` agiscono in modo simile ai dizionari.

```
for valore in insieme:
    print(valore)
# Out: 5
# Out: 6

print(5 in insieme)
# Out: True
print('six' in insieme)
# Out: False
print(len(insieme))
# Out: 2
```

Un insieme può anche essere creato a partire da una sequenza usando la funzione `set()`. Un tipico utilizzo è di rimuovere i duplicati dalle liste, senza però preservarne l'ordine.

```
print( set( ['rosso', 'verde', 'blu', 'rosso'] ) )
# Out: {'rosso', 'blu', 'verde'}
```

Gli insiemi supportano le operazioni di *unione* `|`, *intersezione* `&`, *differenza* `-` e *differenza simmetrica* `^`

```
colori1 = {'rosso', 'verde', 'giallo'}
colori2 = {'rosso', 'blu'}
print(colori1 | colori2)
# Out: {'rosso', 'giallo', 'blu', 'verde'}
print(colori1 - colori2)
# Out: {'giallo', 'verde'}
print(colori1 & colori2)
# Out: {'rosso'}
print(colori1 ^ colori2)
# Out: {'blu', 'giallo', 'verde'}
```

Come esempio, scriviamo una funzione `intersezione_liste(lst, lst2)` che ritorna una lista che contiene i valori comuni alle due liste `lst1` e `lst2`

senza ripetizioni.

```
# crea una lista dall'insieme
def intersezione_liste(lst1, lst2):
    inter = set(lst1) & set(lst2)
    return list(inter)

primi = [2,3,5,7,11,13,17,19,23,29,31,37,41]
fib = [1,1,2,3,5,8,13,21,34,55,89]

print(intersezione_liste(primi, fib))
# Out: [13, 2, 3, 5]

w1 = [ 'quali', 'sono', 'le', 'parole', 'in', 'comune' ]
w2 = [ 'sono', 'le', 'parole', 'che', 'appaiono', 'sia', 'in', 'w1',
'che', 'in', 'w2' ]

print(intersezione_liste(w1, w2))
# Out: ['sono', 'in', 'le', 'parole']
```

## 9.3 TABELLE DI DATI

Un esempio comune di uso di dizionari in Python è la manipolazione di tabelle di dati. Abbiamo visto ad esempio come creare una piccola rubrica associando ad un nome il numero di telefono. Spesso però abbiamo la necessità di memorizzare dati più complessi, come ad esempio memorizzare per ciascuna persona il nome, il telefono e l'anno di nascita. Ad esempio potremmo avere i seguenti dati:

nome	anno	tel
Sofia	1973	5553546
Bruno	1981	5558432
Mario	1992	5555092
Alice	1965	5553546

Un modo comune di memorizzare questi dati è con liste di dizionari. Ogni riga della tabella corrisponde ad un dizionario. Questa collezione di dizionari è poi memorizzata in una lista corrispondente a tutte le righe.

```
colonne = ['nome', 'anno', 'telefono']

dati = [
    { 'nome': 'Sofia', 'anno': 1973, 'tel': '5553546' },
    { 'nome': 'Bruno', 'anno': 1981, 'tel': '5558432' },
    { 'nome': 'Mario', 'anno': 1992, 'tel': '5555092' },
    { 'nome': 'Alice', 'anno': 1965, 'tel': '5553546' },
]

print(dati)
# Out: [{'nome': 'Sofia', 'anno': 1973, 'tel': '5553546'}, {'nome':
'Bruno', 'anno': 1981, 'tel': '5558432'}, {'nome': 'Mario', 'anno':
1992, 'tel': '5555092'}, {'nome': 'Alice', 'anno': 1965, 'tel':
'5553546'}]
```

Si può notare chiaramente come la funzione `print()` non è particolarmente

utile nella stampa di questi dati. Per dati non troppo grandi, utilizzeremo la funzione `pprint()` dall'omonimo modulo.

```
from pprint import pprint
pprint(dati)
# Out: [{'anno': 1973, 'nome': 'Sofia', 'tel': '5553546'},
# Out:  {'anno': 1981, 'nome': 'Bruno', 'tel': '5558432'},
# Out:  {'anno': 1992, 'nome': 'Mario', 'tel': '5555092'},
# Out:  {'anno': 1965, 'nome': 'Alice', 'tel': '5553546'}]
```

## 9.4 ESTRAZIONE DI DATI

Analogamente all'utilizzo del metodo `keys()` nei dizionari, è spesso necessario estrarre singole colonne dai dati e creare dalle tabelle con un sottoinsieme delle colonne esistenti. Possiamo creare funzioni analoghe iterando sulle righe della tabella.

```
def colonna(dati,chiave):  
    '''Ritorna la lista dei valori per la colonna  
    specificata da chiave.'''  
    valori = []  
    for dato in dati:  
        valori.append( dato[chiave] )  
    return valori
```

```
nomi = colonna(dati,'nome')  
pprint(nomi)  
# Out: ['Sofia', 'Bruno', 'Mario', 'Alice']
```

```
def sottotabella(dati,chiavi):  
    '''Ritorna la tabella che include solo le  
    colonne specificate da chiavi.'''  
    ndati = []  
    for dato in dati:  
        ndato = {}  
        for chiave in chiavi:  
            ndato[chiave] = dato[chiave]  
        ndati.append( ndato )  
    return ndati
```

```
pprint(sottotabella(dati,['nome','anno']))  
# Out: [{'anno': 1973, 'nome': 'Sofia'},  
# Out:  {'anno': 1981, 'nome': 'Bruno'},  
# Out:  {'anno': 1992, 'nome': 'Mario'},  
# Out:  {'anno': 1965, 'nome': 'Alice'}]
```

## 9.5 RICERCA DI DATI

Vorremmo ora accedere ai dati per, ad esempio, trovare il numero di telefono corrispondente ad un nome. Con i dati in questo formato, possiamo iterare sulle lista e accedere ai singoli dizionari per fare operazioni arbitrarie. Ad esempio, scriviamo una funzione che ritorna il valore di una chiave per un determinato nome. In caso il nome non è presente utilizzeremo il valore speciale `None`.

```
def ricerca(dati,nome,chiave):
    for dato in dati:
        if dato['nome'] == nome:
            return dato[chiave]
    return None

print(ricerca(dati,'Mario','tel'))
# Out: 5555092
```

Questa operazione è così comune che spesso è utile creare degli *indici* che permettono di accedere direttamente alle righe della tabella. Ad esempio, potremmo creare un dizionario che associa i nomi alle righe della tabella.

```
indice = {}
for numero_riga, dato in enumerate(dati):
    indice[dato['nome']] = numero_riga

def ricerca_con_indice(dati,indice,nome,chiave):
    if nome in indice:
        numero_riga = indice[nome]
        dato = dati[numero_riga]
        return dato[chiave]
    else:
        return None

print(ricerca_con_indice(dati,indice,'Mario','tel'))
# Out: 5555092
```

## 9.6 ORDINAMENTO DI DATI

Abbiamo visto che la funzione `sorted()` permette di ordinare liste di valori semplici, come interi o stringhe. È spesso utile ordinare tabelle di dati, che però possono essere ordinati rispetto a ciascuna colonna. Ad esempio, potremmo ordinare per nome o anno di nascita. Il parametro opzionale `key` della funzione `sorted()` ci permette di fare esattamente questo. Il parametro `key` si aspetta il nome di una funzione che ritorna il valore da usare per l'ordinamento. Per ordinare in modo inverso, basta usare il parametro opzionale `reverse`.

```
# dati non ordinati
pprint(dati)
# Out: [{'anno': 1973, 'nome': 'Sofia', 'tel': '5553546'},
# Out:  {'anno': 1981, 'nome': 'Bruno', 'tel': '5558432'},
# Out:  {'anno': 1992, 'nome': 'Mario', 'tel': '5555092'},
# Out:  {'anno': 1965, 'nome': 'Alice', 'tel': '5553546'}]

# dati ordinati per nome
def nome_dato(dato): return dato['nome']
pprint(sorted(dati, key=nome_dato))
# Out: [{'anno': 1965, 'nome': 'Alice', 'tel': '5553546'},
# Out:  {'anno': 1981, 'nome': 'Bruno', 'tel': '5558432'},
# Out:  {'anno': 1992, 'nome': 'Mario', 'tel': '5555092'},
# Out:  {'anno': 1973, 'nome': 'Sofia', 'tel': '5553546'}]

# dati ordinati per anno di nascita
def anno_dato(dato): return dato['anno']
pprint(sorted(dati, key=anno_dato))
# Out: [{'anno': 1965, 'nome': 'Alice', 'tel': '5553546'},
# Out:  {'anno': 1973, 'nome': 'Sofia', 'tel': '5553546'},
# Out:  {'anno': 1981, 'nome': 'Bruno', 'tel': '5558432'},
# Out:  {'anno': 1992, 'nome': 'Mario', 'tel': '5555092'}]

# dati ordinati per anno di nascita inverso
pprint(sorted(dati, key=anno_dato, reverse=True))
# Out: [{'anno': 1992, 'nome': 'Mario', 'tel': '5555092'},
```

```
# Out: {'anno': 1981, 'nome': 'Bruno', 'tel': '5558432'},  
# Out: {'anno': 1973, 'nome': 'Sofia', 'tel': '5553546'},  
# Out: {'anno': 1965, 'nome': 'Alice', 'tel': '5553546'}]
```

## 10 ACCESSO AI DATI

I dati elaborati dai programmi sono mantenuti in memorie secondarie, come dischi e memorie flash, o reperibili attraverso connessioni di rete, come le risorse su Internet. Python permette di accedere a queste due tipologie di memorizzazione in modo semplice, astruendo le complessità dei sistemi di memorizzazione e di connessione alla rete. In questo capitolo, descriveremo come accedere a informazioni sia su disco che su internet.

## 10.1 FILE E PERCORSI

I file mantengono nella memoria di un computer, generalmente su disco, informazioni di qualsiasi tipo: testi, dati, tabelle, immagini, video, musica, ecc. In Python si accede ai file con oggetti predefiniti di tipo `file`. Un oggetto di tipo `file`, rappresenta uno specifico file che può essere letto, scritto o modificato tramite i metodi del tipo `file`. Come vedremo anche documenti che non sono su disco ma sono accessibili tramite reti di comunicazione sono gestiti da Python in modo simile ai file su disco.

Se i file a cui vogliamo accedere non sono nella cartella da cui abbiamo lanciato l'interprete, dobbiamo indicarne il percorso. In questo libro non andremo in dettaglio sulla definizione dei percorsi e il loro utilizzo, menzionando solo un piccolo riassunto dei casi più comuni.

Il percorso è formato dai nomi delle directory che contengono il file separati da un carattere separatore. Su Linux e Mac OS X, il carattere separatore è lo slash `/`, su Windows è tradizionalmente il backslash `\`, anche se oggi Windows accetta anche `/`. Ad esempio usiamo `docs/myfile.txt` per accedere al file `myfile.txt` nella directory `docs`. Per indicare la cartella precedente nella gerarchia si può usare il nome speciale `..`. Ad esempio `../myfile.txt` indica il file nella cartella precedente.

Un percorso assoluto determina la locazione del file partendo dalla root directory. Per Linux e Mac OS X la root directory è indicata con `/`. Ad esempio `/home/user/Documents/myfile.txt`. Su Windows si deve anche indicare il volume (o disco). Ad esempio `C:\Users\user\Documents\myfile.txt`. L'utilizzo di percorsi globali è generalmente sconsigliato perché non permette al programma di funzionare su file diversi o di essere portato su altre macchine.

## 10.2 APERTURA DI FILE

Prima di poter accedere ad un file bisogna ottenere un oggetto di tipo `file` che lo rappresenta, usando la funzione predefinita `open(name, mode='rt', ...)`. Questa prende come primo argomento obbligatorio il nome del file o il percorso che si vuole aprire, ad esempio `'myfile.txt'`.

Il secondo argomento, che è opzionale, indica la *modalità di apertura* che può assumere vari valori. I più comuni sono `'r'` per aprire in lettura, `'w'` per aprire in scrittura, e `'a'` per aggiungere alla fine del file. Di default la modalità è di lettura. A seguire si può indicare il tipo di file che può essere `t` per un file di testo e `b` per un file binario.

Se l'apertura del file va a buon fine, si potrà elaborare il contenuto del file usando i metodi del tipo `file`. Se invece si tenta di aprire in lettura un file che non esiste si ottiene un errore. Quando si è terminato di usare il file, è bene chiuderlo, tramite il metodo `close()`.

```
f = open('alice.txt')
# qualcosa col file ...
f.close()

f = open('non_esiste.txt')
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: FileNotFoundError: [Errno 2] No such file or directory:
'non_esiste.txt'
```

Quando si lavora con i file è preferibile usare il costrutto `with` che gestisce in modo automatico la chiusura del file, anche se si dovesse verificare un errore durante l'elaborazione.

```
with open(...) as f:
    istruzioni
```

Nel blocco di istruzioni del `with`, il file è aperto e il relativo oggetto è il valore della variabile `f`. Non appena l'esecuzione esce dal blocco il file è automaticamente chiuso anche se si esce a causa di un errore.

```
with open('alice.txt') as f:
    print('aperto')
    # fai qualcosa col file ...
# Out: aperto
```

Se del file non specifichiamo il percorso, o il percorso è locale, il file è cercato nella cartella detta *current working directory*, o più brevemente *cwd*, che è accessibile tramite la funzione `getcwd()` del modulo `os`. Questa cartella è quella da cui abbiamo eseguito l'interprete.

```
import os

print(os.getcwd())
# Out: /Users/fabio/Documents/Work/books/fondamentibook/ch10
```

## 10.3 CODIFICA DEI FILE DI TESTO

Sebbene abbiamo visto come le stringhe in Python possono rappresentare qualunque alfabeto, rimane in problema di come salvare il testo su disco. Sfortunatamente ci sono varie codifiche, incompatibili tra loro, adatte a questo scopo. Il default su Mac e Linux è la codifica Unicode UTF8. Su Windows non c'è un default standard. Quando si apre un file di testo in Python, l'interprete adotta la codifica di default del sistema operativo. Possiamo però forzare una codifica diversa utilizzando il parametro opzionale `encoding` della funzione `open`. Per gli esempi di questo libro, utilizzeremo l'UTF8 se non indicato in modo esplicito.

Un'ulteriore differenza tra sistemi operativi, e quindi i file creati sugli stessi, è il carattere utilizzato per andare a capo che è `\n` su Mac e Linux e `\r\n` su Windows. In questo caso, Python converte i vari formati internamente in modo che il testo caricato sia valido indipendentemente dalla codifica.

## 10.4 LETTURA DI FILE

Come file d'esempio usiamo *Alice's Adventures in Wonderland* di Lewis Carroll che si può scaricare da [Project Gutenberg](#). Utilizzeremo la versione in testo UTF8. Dopo avere aperto un file in lettura, possiamo leggerne il contenuto con il metodo `read()`.

```
f = open('alice.txt')

testo = f.read()
print(len(testo))
# Out: 163781

print(testo[686:1020])
# Out: CHAPTER I. Down the Rabbit-Hole
# Out:
# Out: Alice was beginning to get very tired of sitting by her
# Out: sister on the
# Out: bank, and of having nothing to do: once or twice she had
# Out: peeped into the
# Out: book her sister was reading, but it had no pictures or
# Out: conversations in
# Out: it, 'and what is the use of a book,' thought Alice 'without
# Out: pictures or
# Out: conversations?
```

Se proviamo a chiamare ancora il metodo `read()` otteniamo la stringa vuota, perché la prima chiamata ha letto l'intero file e non è rimasto nessun carattere da leggere.

```
print(f.read())
# Out:
f.close()
```

Possiamo utilizzare il metodo `read()` anche con l'istruzione `with`.

```
with open('alice.txt') as f:
    testo = f.read()
    print(len(testo))
# Out: 163781
```

Il metodo `read()` può prendere un parametro opzionale che determina il numero massimo di caratteri da leggere. Il metodo `readline()` permette di leggere una linea alla volta. Per provarlo dobbiamo riaprire il file, avendolo già chiuso in precedenza. Facciamo questo esempio con l'istruzione `with`.

```
with open('alice.txt') as f:
    first = f.readline()
    print(repr(first))
# Out: "\uffffProject Gutenberg's Alice's Adventures in Wonderland,
by Lewis Carroll\n"
```

Il codice alla riga precedente dà problemi, dato che l'encoding dei file di testo scelto da Python è quello di default del sistema, ma Project Gutenberg usa il formato `utf-8-sig` che è una variante non compatibile del formato standard UTF8. Per assicurarci di leggere i file correttamente possiamo specificare l'encoding in modo diretto.

```
with open('alice.txt', encoding='utf-8-sig') as f:
    print(repr(f.readline()))
# Out: "Project Gutenberg's Alice's Adventures in Wonderland, by
Lewis Carroll\n"
```

Avendo determinato l'encoding corretto, possiamo anche usare libri in altre lingue, ad esempio la versione tradotta in italiano del libro.

```
with open('alice_it.txt', encoding='utf-8-sig') as f:
    print(repr(f.readline()))
# Out: "The Project Gutenberg eBook of Le avventure d'Alice nel
paese delle\n"
```

Il metodo `readlines()` legge tutte le linee fino alla fine del file. Questo ci dà un modo semplice per contare le linee di testo di lunghi documenti,

accedere a linee specifiche, o iterare sulle linee di un file.

```
with open('alice.txt', encoding='utf-8-sig') as f:
    linee = f.readlines()
    print(len(linee))
    print(linee[200])
    print(linee[400])
# Out: 3735
# Out: she felt a little nervous about this; 'for it might end, you
know,' said
# Out:
# Out: anything had happened.) So she began again: 'Ou est ma
chatte?' which
# Out:
```

Gli oggetti di tipo `file` possono anche essere iterati come se fossero una sequenza di linee direttamente. Il vantaggio di questo metodo è che il contenuto del file non deve risiedere completamente in memoria. Come esempio, scriviamo una funzione che ritorna la lista degli indici di riga in cui appare una stringa. Questo funzione simula la ricerca nei documenti che usiamo spesso negli editor di testo.

```
def ricerca_linee(nome_file, encoding, stringa):
    '''Ritorna la lista dei numeri delle linee del
file nome_file in cui appare la stringa.'''
    with open(nome_file, encoding=encoding) as f:
        lista_indici = []
        indice_corrente = 1
        for linea in f:
            if linea.find(stringa) != -1:
                lista_indici.append(indice_corrente)
                indice_corrente += 1
        return lista_indici

indici = ricerca_linee('alice.txt', 'utf-8-sig', 'Turtle')
print(indici)
# Out: [2213, 2353, 2355, 2357, 2371, 2389, 2396, 2402, 2409, 2410,
2414, 2419, 2421, 2425, 2431, 2438, 2441, 2448, 2452, 2456, 2463,
2468, 2483, 2485, 2493, 2499, 2508, 2520, 2532, 2536, 2550, 2559,
2563, 2571, 2577, 2583, 2587, 2594, 2626, 2632, 2638, 2640, 2679,
```

```
2684, 2689, 2696, 2707, 2712, 2740, 2746, 2751, 2774, 2782, 2784,  
2786, 2789, 2812, 3347, 3357]  
indici = ricerca_linee('alice.txt', 'utf-8-sig', 'Alice')  
print(len(indici))  
# Out: 396
```

## 10.5 SCRITTURA DI FILE

Si possono scrivere dati su file aprendo il file in scrittura con modalità `'w'` o `'a'`. In modalità `'w'` il file è creato se non esiste già, o il contenuto precedente è sovrascritto. Per scrivere in un file si può usare il metodo `write()` passando una stringa di testo da scrivere. `write()` ritorna il numero di caratteri scritti. Possiamo verificare l'avvenuta scrittura del file leggendone in contenuto di nuovo.

```
with open('testo.txt', 'w') as f:
    f.write('Questo è il contenuto del file.')
# Out: 31

with open('testo.txt') as f:
    print(f.read())
# Out: Questo è il contenuto del file.
```

Il metodo `writelines()` permette di scrivere una lista di stringhe. Se si vuole ad esempio scrivere una lista di stringhe come righe del file basta che ogni linea contenga esplicitamente il carattere di fine linea `'\n'`.

```
with open('lista.txt', 'w') as f:
    f.writelines(['Linea1\n', 'Linea2\n'])

with open('lista.txt') as f:
    print(f.read())
# Out: Linea1
# Out: Linea2
# Out:
```

In modalità *append* `'a'` le scritture avvengono in coda all'attuale contenuto. Ad esempio, se si vuole mantenere un file di log, cioè un file in cui vengono registrate azioni o eventi, è naturale aggiornarlo in questa modalità, aggiungendo così di volta in volta le nuove registrazioni e mantenendo quelle precedenti.

```
with open('append.txt', 'w') as f:  
    f.write('Linea1.\n')  
# Out: 8
```

```
with open('append.txt', 'a') as f:  
    f.write('Linea2.\n')  
# Out: 8
```

```
with open('append.txt') as f:  
    print(f.read())  
# Out: Linea1.  
# Out: Linea2.  
# Out:
```

## 10.6 INPUT E OUTPUT DI DATI

Mentre il testo può essere salvato direttamente, altri tipi di informazioni devono essere codificate esplicitamente in formati appropriati. Se vogliamo salvare dati arbitrari in Python, il formato più semplice da usare è il **JSON** che permette di codificare in modo standard qualunque combinazione di liste, dizionari, numeri, booleani, stringhe e `None`. JSON è particolarmente utile perché supportato dalla maggioranza di applicazioni su Internet. In Python, il formato JSON è implementato dal modulo `json` con le funzioni `dump()` per salvare i dati e `load()` per leggerli.

```
import json

# definisce i dati
dati = [
    { 'nome': 'Sofia', 'anno': 1973, 'tel': '5553546' },
    { 'nome': 'Bruno', 'anno': 1981, 'tel': '5558432' },
    { 'nome': 'Mario', 'anno': 1992, 'tel': '5555092' },
    { 'nome': 'Alice', 'anno': 1965, 'tel': '5553546' },
]

# salva i dati su disco
with open('dati.json', 'w') as f:
    json.dump(dati, f)

# per verificare il salvataggio, rileggiamo i dati
with open('dati.json') as f:
    nuovi_dati = json.load(f)
print(dati == nuovi_dati)
# Out: True
```

## 10.7 ACCEDERE A DOCUMENTI SUL WEB

In Python la manipolazione di dati accessibili tramite connessioni remote, come ad esempio Internet, usa oggetti molto simili ai file. In questo caso possiamo solo leggere documenti dalla rete e non salvare file su Web. La differenza principale rispetto ai file è nell'apertura, perché i dati in remoto hanno locazioni che sono diverse dai percorsi locali e il loro accesso può essere regolato da autenticazioni o cookies. In questo capitolo vedremo solamente gli accessi più semplici ma ci ritorneremo più avanti.

Sul Web le risorse sono localizzate tramite URL o *Uniform Resource Locator*. Un esempio di URL è

```
http://en.wikipedia.org/wiki/Uniform_resource_locator
```

Lo URL è simile al percorso di un file ed è composto da tre parti principali. Lo *schema* `http://` è il protocollo usato per l'accesso, in questo caso HTTP o *HyperText Transfer Protocol*. Il nome del *dominio* `en.wikipedia.org` è il nome della macchina, virtuale o reale, dove la risorsa è localizzabile. Infine, il *percorso* della risorsa `/wiki/Uniform_resource_locator` identifica la risorsa nel dominio. L'URL diventa più complicato per siti dove l'utente può interagire con il sito. Ad esempio su Google l'URL contiene anche i parametri della ricerca. Per ora consideriamo solo gli URL più semplici.

Per accedere a URL, Python mette a disposizione la libreria `urllib` costituita da vari moduli. Il modulo `urllib.request` permette di accedere a risorse URL in lettura. Per scaricare una pagina dal sito di Python, si può usare la funzione `urlopen()` che ritorna un'oggetto simile ad un file. Quest'ultimo può essere usato con la funzione `read()` e in congiunzione all'istruzione `with`. In questi casi, i dati sono letti sempre come binario. Per convertirli in testo dobbiamo chiamare esplicitamente il metodo `decode()` sui dati di ritorno.

```
from urllib.request import urlopen
```

```
# apre una pagina web
with urlopen('http://python.org') as f:
    page = f.read()

# dati in binario
print(page[:50])
# Out: b'<!doctype html>\n<!--[if lt IE 7]> <html class="n'

# dati come testo
page = page.decode('utf8')
print(page[:50])
# Out: <!doctype html>
# Out: <!--[if lt IE 7]> <html class="n
```

Come altro esempio, possiamo scaricare un'immagine da Wikipedia e salvarla su disco, badando bene a salvare il formato binario con `'wb'`.

```
url = ('https://upload.wikimedia.org/wikipedia/' +
      'commons/thumb/d/df/Face-plain.svg/' +
      '200px-Face-plain.svg.png')
with urlopen(url) as f:
    img = f.read()

with open('face.png', 'wb') as f:
    f.write(img)
# Out: 19262
```



## 11 ELABORAZIONE DI TESTO

L'elaborazione del testo è una problematica molto comune in programmazione con usi disparati come l'editing di documenti, il controllo ortografico, la ricerca e l'estrazione di informazioni, ecc. Per questo Python ha dotato le stringhe di svariati metodi che introdurremo in questo capitolo per risolvere il problema della ricerca nei documenti. Quest'ultima si può formalizzare come il problema di determinare il file, in una collezione di files, che è più attinente ad una lista di parole di ricerca. Questo è uno degli ingredienti fondamentali dei search engines, come Google o Bing. Considereremo una versione estremamente semplificata del problema che però ne mostra gli elementi principali.

## 11.1 METODI DELLE STRINGHE

La manipolazione del testo si può eseguire scrivendo funzioni che accedono direttamente ai singoli caratteri. In questo modo si possono implementare tutte le possibili manipolazioni, ma il codice risultante è complesso e pronò ad errori. In Python, è più comune elaborare il testo combinando appropriatamente i metodi delle stringhe che permettono di esprimere operazioni complesse in modo succinto.

I metodi `lower()` e `upper()` ritornano una copia della stringa in cui tutti i caratteri alfabetici sono stati trasformati, rispettivamente, in minuscolo e in maiuscolo. Il metodo `count()` conta il numero di occorrenze di una sottostringa nella stringa. Il metodo può prendere anche fino a due parametri opzionali che restringono la ricerca delle occorrenze a una porzione della stringa.

```
s = 'Il Numero 1000'
print(s.lower())
# Out: il numero 1000
print(s.upper())
# Out: IL NUMERO 1000

a = "Ma che bella giornata."
print(a.count('e'))
# Out: 2
print(a.count('z'))
# Out: 0
print(a.count('giornata'))
# Out: 1

# conta dalla posizione 9 in poi
print(a.count('i', 9))
# Out: 1
# dalla posizione 6 alla 9 esclusa
print(a.count('i', 6, 9))
# Out: 0
```

Per illustrare l'uso di questi metodi scriviamo una funzione `conta_vocali()` che conta il numero di vocali non accentate, minuscole e maiuscole, presenti nella stringa di input.

```
def conta_vocali(s):
    '''Conta le vocali non accentate in s.'''
    # Per contare anche le vocali maiuscole
    s = s.lower()
    count = 0
    for v in 'aeiou':
        count += s.count(v)
    return count

print(conta_vocali("che bello andare a spasso"))
# Out: 9
print(conta_vocali("Nn c sn vcl"))
# Out: 0
```

Il metodo `find()` ritorna la posizione in cui inizia la prima occorrenza di una stringa, se non ci sono occorrenze ritorna `-1`.

```
s = 'che bello andare a spasso'
print(s.find('bello'))
# Out: 4
# cerca a partire dalla posizione 4
print(s.find('e', 4))
# Out: 5
# cerca tra le posizioni 10 e 20 esclusa
print(s.find('bello', 10, 20))
# Out: -1
```

Il metodo `splitlines()` ritorna la lista delle linee della stringa. Se specifichiamo il parametro opzionale `True` anche i caratteri di fine linea sono ritornati.

```
testo = '''Prima linea,
seconda linea
e terza linea.'''
print(testo.splitlines())
```

```
# Out: ['Prima linea,', 'seconda linea', 'e terza linea.']
print(testo.splitlines(True))
# Out: ['Prima linea,\n', 'seconda linea\n', 'e terza linea.']
```

Un altro metodo molto utile delle stringhe è `split()` che ritorna la lista delle sottostringhe che sono separate dai caratteri spazio o da una stringa passata come parametro. Si faccia attenzione che, in generale, `split()` non è equivalente a `split(' ')` dato che nel primo caso le sottostringhe vuote vengono soppresse, mentre nel secondo restano presenti.

```
s = "Una frase d'esempio, non    troppo lunga"
# il separatore di default è lo spazio
print(s.split())
# Out: ['Una', 'frase', "d'esempio,", 'non', 'troppo', 'lunga']

# altri separatori
print(s.split(','))
# Out: ["Una frase d'esempio", ' non    troppo lunga']
print(s.split('p'))
# Out: ["Una frase d'esem", 'io, non    tro', '', 'o lunga']
print(s.split('pp'))
# Out: ["Una frase d'esempio, non    tro", 'o lunga']

# fa al massimo 2 separazioni
print(s.split('p', 2))
# Out: ["Una frase d'esem", 'io, non    tro', 'po lunga']

# differenza per le stringhe ripetute
print(s.split(' '))
# Out: ['Una', 'frase', "d'esempio,", 'non', '', '', '', 'troppo', 'lunga']
```

A volte può essere utile eliminare spazi sono all'inizio e alla fine di una stringa, ad esempio per normalizzare un testo di input, eliminare i fine linea dopo `splitlines()` o determinare le parole con `split()`. Per fare questo usiamo il metodo `strip()`.

```
s = ' spazio prima e dopo '
print(repr(s))
```

```
# Out: ' spazio prima e dopo '  
print(repr(s.strip()))  
# Out: 'spazio prima e dopo'
```

Il metodo `replace()` sostituisce tutte le occorrenze di una data sottostringa con un'altra sottostringa, rispettando la differenza tra maiuscole e minuscole. `replace()` può anche essere usato per cancellare parte di una stringa.

```
s = "Ciao Bruno come stai?"  
print(s.replace('Bruno', 'Sara'))  
# Out: Ciao Sara come stai?  
print(s.replace('bruno', 'sara'))  
# Out: Ciao Bruno come stai?  
print(s.replace('come ', '').replace('?', ' bene?'))  
# Out: Ciao Bruno stai bene?
```

Per creare stringhe formattate a partire dai valori si usa il metodo `format()`. La stringa su cui si chiama il metodo esprime il formato in cui convertire i valori passati al metodo. Il formato si riferisce ai valori con `{i}`, che indica l'*i*-esimo argomento del metodo, e `{var}`, che indica l'argomento chiave `var=`. L'indice può essere omesso dal formato se si accede ai valori in modo continuo. La stringa formato ha molte altre opzioni che non elencheremo esplicitamente, riferendo il lettore alla documentazione di Python.

```
'{} per {} uguale {}'.format(5,3,5*3)  
# Out: '5 per 3 uguale 15'  
'{nome} nato in {indirizzo} nel {anno}'.format(  
    nome='Mario', indirizzo='Italia', anno='1974')  
# Out: 'Mario nato in Italia nel 1974'
```

## 11.2 ELABORAZIONE DEL TESTO

Vediamo ora un esempio che usa i metodi appena descritti. Abbiamo visto in precedenza come elaborare informazioni utilizzando tabelle di dati. Spesso però i dati sono contenuti in stringhe da cui devono essere estratti i singoli valori. Siamo interessati a scrivere una funzione `rubrica(elenco, nome)` che prende in input una stringa `elenco` che in ogni linea contiene un nome, il carattere `:` e poi un numero di telefono. La funzione deve ritornare il numero di telefono corrispondente al nome `nome` preso anch'esso come input. Procederemo riducendo prima le stringhe `nome` e `elenco` in minuscole, con il metodo `lower()`, per far sì che poi la ricerca del nome non sia intralciata da differenze tra maiuscole e minuscole. Poi suddividiamo `elenco` in linee, tramite il metodo `splitlines()` e ogni linea in nome e numero con `split()`. Infine eliminiamo gli spazi a destra e sinistra di ogni elemento con il metodo `strip()`.

```
def ricerca(elenco, nome):
    '''Ritorna il numero di telefono nell'elenco
    per la riga con nome nome.'''
    nome = nome.lower()
    elenco = elenco.lower().splitlines()
    for e in elenco:
        e_nome, e_num = e.split(':')
        if nome == e_nome.strip():
            return e_num.strip()
    return 'Non esiste'

elenco = '''Marco: 5551234
Luisa: 5557653
Sara: 5558723'''

print(ricerca(elenco, 'Marco'))
# Out: 5551234
print(ricerca(elenco, 'Luisa'))
# Out: 5557653
print(ricerca(elenco, 'Giuseppe'))
```

```
# Out: Non esiste
```

Possiamo anche convertire la stringa di inizio in un dizionario, usando una funzione simile alla precedente.

```
def rubrica_to_dict(elenco):
    '''Converte un elenco da testo a tabella.'''
    d = {}
    elenco = elenco.lower().splitlines()
    for e in elenco:
        nome, numero = e.split(':')
        d[nome.strip()] = numero.strip()
    return d

elenco = '''Marco: 5551234
Luisa: 5557653
Sara: 5558723'''

dizionario = rubrica_to_dict(elenco)

from pprint import pprint
pprint(dizionario)
# Out: {'luisa': '5557653', 'marco': '5551234', 'sara': '5558723'}
```

Scriviamo una funzione `camel(s)` che prende in input una stringa `s` del tipo `fraseSenzaSpazi` e ritorna la lista delle parole, che nel nostro esempio è `['frase', 'senza', 'spazi']`. Questo formato è molto usato da programmatori come convenzioni per i nomi delle variabili. Lo decodifichiamo usando `replace()` per sostituire ogni carattere maiuscolo con il corrispondente minuscolo preceduto da spazi, seguito da `split()` per separare le parole.

```
def camel(s):
    for c in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
        s = s.replace(c, " "+c.lower())
    return s.split()

print(camel('fraseSenzaSpazi'))
# Out: ['frase', 'senza', 'spazi']
```

```
print(camel('sentenceWithoutSpaces'))
# Out: ['sentence', 'without', 'spaces']
```

È spesso molto utile creare una lista delle parole contenute in una stringa. Per parola intendiamo una qualsiasi sequenza di caratteri alfabetici, maiuscoli o minuscoli, di lunghezza massimale. Per prima cosa vediamo come determinare i caratteri non alfabetici presenti in una stringa. Possiamo usare il metodo `isalpha()` delle stringhe che ritorna `True` se la stringa a cui è applicato contiene solamente caratteri alfabetici. Scriviamo allora una funzione `noalpha(s)` che ritorna una stringa contenente tutti i caratteri non alfabetici in `s`, senza ripetizioni. Scritta in questo modo, `noalpha()` funzionerà per tutte le lingua possibili.

```
def noalpha(s):
    '''Ritorna una stringa contenente tutti i
    caratteri non alfabetici contenuti in s,
    senza ripetizioni'''
    noa = ''
    for c in s:
        if not c.isalpha() and c not in noa:
            noa += c
    return noa

print(noalpha("Frase con numeri 0987"))
# Out: 0987
print(noalpha("Frase con simboli vari [],{} %&#@"))
# Out: [],{}%&#@
print(noalpha("FraseSenzaCaratteriNonAlfabetici"))
# Out:
```

Per determinare la lista delle parole possiamo usare il metodo `split()` dopo aver sostituito convertito il testo in minuscolo con `lower()` e sostituito tutti i caratteri non alfabetici con il carattere spazio tramite `replace()`.

```
def words(s):
    '''Ritorna la lista delle parole contenute
    nella stringa s'''
    noa = noalpha(s)
    for c in noa:
```

```
        s = s.replace(c, ' ')
    return s.lower().split()

print(words("Che bel tempo, usciamo!"))
# Out: ['che', 'bel', 'tempo', 'usciamo']
```

Per gli esempi di seguito utilizzeremo file da libri scaricati da [Project Gutenberg](#). I metodi sviluppati funzioneranno su qualunque libro. Per generare i nostri risultati, abbiamo utilizzato le versioni in testo UTF8 del libro *Alice in Wonderland* di Lewis Carroll.

Se vogliamo creare una lista di tutte le parole in un file di testo possiamo sfruttare la funzione `words()` che abbiamo definito. Ad esempio possiamo contare il numero di parole in *Alice in Wonderland* sia in versione inglese che in versione italiana. Notiamo come i libri sono approssimativamente di lunghezza uguale, come ci si aspetta.

```
def fwords(fname,encoding):
    with open(fname, encoding=encoding) as f:
        testo = f.read()
    return words(testo)

parole = fwords('alice.txt','utf-8-sig')
print(len(parole))
# Out: 30419
print(parole[1000:1005])
# Out: ['bats', 'eat', 'cats', 'for', 'you']

parole_italiano = fwords('alice_it.txt','utf-8-sig')
print(len(parole_italiano))
# Out: 27794
print(parole_italiano[1000:1005])
# Out: ['era', 'in', 'fondo', 'e', 'andando']
```

La lista prodotta contiene tutte le occorrenze delle parole con le appropriate ripetizioni. Possiamo rimuovere le ripetizioni usando `set()`. Da qui possiamo notare come la versione italiana utilizzi un vocabolario più ampio, probabilmente dovuto alla differenze grammaticali e alla coniugazioni dei verbi.

```
parole_uniche = set(fwords('alice.txt','utf-8-sig'))  
print(len(parole_uniche))  
# Out: 3007
```

```
parole_uniche_italiano= set(fwords('alice_it.txt','utf-8-sig'))  
print(len(parole_uniche_italiano))  
# Out: 5180
```

## 11.3 RICERCA DI DOCUMENTI

Abbiamo ora gli strumenti per risolvere agevolmente il nostro problema iniziale: data una collezione di documenti e una lista di parole, trovare il documento che è più attinente alla lista di parole. L'idea è di calcolare la frequenza con cui le parole della lista occorrono in ogni documento e poi scegliere il documento in cui occorrono con maggiore frequenza. Per prima cosa dobbiamo, per ogni documento, contare le occorrenze di ogni parola della lista. Un dizionario è perfetto per mantenere un conteggio, infatti basterà creare un dizionario le cui chiavi sono le parole della lista e ad ogni parola è associato il conteggio delle occorrenze della parola. Schematicamente, procederemo come segue.

1. per ogni documento, creiamo la lista delle parole che appaiono nel documento;
2. creiamo un dizionario per mantenere le frequenze calcolando, per ogni parola nella lista di ricerca, il numero delle sue occorrenze nelle parole del documento, e aggiungendo la coppia parola-occorrenze al dizionario;
3. per tenere conto che i documenti possono avere lunghezze diverse, modifichiamo il dizionario normalizzando le frequenze assolute in frequenze relative, cioè dividendo per il numero totale di parole;
4. infine, assegniamo uno *score* ad ogni documento pari alla somma delle frequenze percentuali delle parole nella lista data: più alto è lo score e più attinente è ritenuto il documento.

Conviene decomporre la procedura in due funzioni. La prima ritorna il dizionario delle frequenze relativo ad un file e alla lista di parole da ricercare. La seconda funzione ritorna un dizionario che per ogni file, in una lista specificate, associa il suo score relativamente alla parole da ricercare. Decomponendo il problema in questo modo, si ha anche la possibilità di fare analisi aggiuntive sui dizionari ritornati.

```

def wfreq(fname, ricerca, enc):
    '''Ritorna un dizionario che ad ogni parola nella
    lista ricerca associa la sua frequenza
    percentuale nel file fname. Il file è
    decodificato tramite la codifica enc.'''

    # ottiene la lista delle parole
    parole = fwords(fname, enc)
    # prepara il dizionario delle frequenze
    frequenze = {}
    # per ogni parole nella ricerca
    for parola in ricerca:
        # calcola le occorrenze
        occ = parole.count(parola.lower())
        # calcola frequenza percentuale
        freq = occ*100/len(parole)
        # aggiorna il dizionario
        frequenze[parola] = round(freq,3)
    return frequenze

fname = 'alice.txt'
ricerca = ['alice', 'rabbit', 'turtle', 'king']
freq = wfreq(fname, ricerca, 'utf-8-sig')
print(freq)
# Out: {'king': 0.207, 'alice': 1.325, 'turtle': 0.194, 'rabbit':
0.168}

```

Quindi la parola “alice” ha una frequenza percentuale superiore all’1%, cioè più di una parola su 100 è la parola “alice”, mentre “turtle” appare con una frequenza inferiore allo 0.2%, cioè appare meno di una volta ogni 500 parole. Passiamo ora alla seconda funzione.

```

def scores(fnames, ricerca, enc):
    '''Ritorna un dizionario che ad ogni nome di file
    in fnames associa il suo punteggio relativamente
    alla lista di parole ricerca. I file sono
    decodificati tramite la codifica enc.'''
    frequenze = {}
    for fname in fnames:
        # dizionario delle frequenze di fname
        f = wfreq(fname, ricerca, enc)

```

```

# score arrotondata
frequenze[fname] = round(sum(f.values()), 3)
return frequenze

```

Proviamo la funzione su una lista di file che contengono alcuni libri celebri ottenuti da Project Gutenberg: *Frankeinstein* di Mary Shelley, *Il Principe* di Nicolò Machiavelli (versione inglese), *Moby Dick* di Herman Melville, *Treasure Island* di Robert Stevenson e *The Adventures of Sherlock Holmes* di Arthur Conan Doyle.

```

fnames = [ 'alice.txt', 'holmes.txt',
           'frankenstein.txt', 'prince.txt',
           'mobydick.txt', 'treasure.txt' ]

ricerca = [ 'monster', 'horror', 'night' ]
pprint(scores(fnames, ricerca, 'utf-8-sig'))
# Out: {'alice.txt': 0.016,
# Out:  'frankenstein.txt': 0.216,
# Out:  'holmes.txt': 0.119,
# Out:  'mobydick.txt': 0.103,
# Out:  'prince.txt': 0.023,
# Out:  'treasure.txt': 0.066}

```

Per terminare il nostro piccolo search engine aggiungiamo una funzione che ritorna la lista dei documenti in ordine crescente di score. Utilizzeremo la funzione `sorted()` applicata alla sequenza della coppie (chiave, valore) ritornata da `scores().items()` e ordinando in modo inverso con `reverse`.

```

def extract_value(kv): return kv[1]
def searchdocument(fnames, ricerca, enc):
    '''Ritorna la lista ordinata per score dei
    documenti in fnames per le parole in ricerca.'''
    s = scores(fnames, ricerca, enc)
    return sorted(s.items(), key=extract_value,
                  reverse=True)

ricerca = [ 'monster', 'horror', 'night' ]
pprint(searchdocument(fnames, ricerca, 'utf-8-sig'))
# Out: [('frankenstein.txt', 0.216),

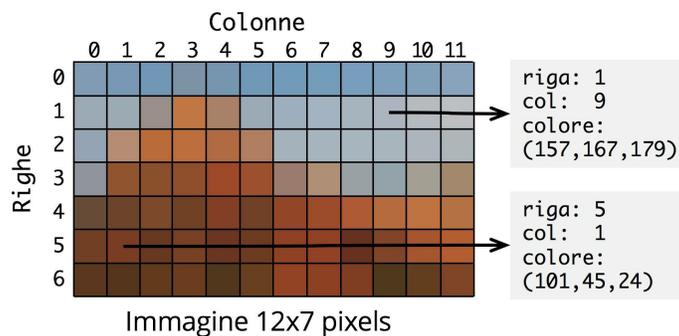
```

```
# Out: ('holmes.txt', 0.119),  
# Out: ('mobydick.txt', 0.103),  
# Out: ('treasure.txt', 0.066),  
# Out: ('prince.txt', 0.023),  
# Out: ('alice.txt', 0.016)]
```

# 12

## ELABORAZIONE DI IMMAGINI

In questo capitolo implementeremo semplici operazioni su immagini digitali, che sono rappresentazioni al computer di fotografie e grafica. In generale, un'immagine digitale è rappresentata come una matrice di colori. Ogni elemento della matrice si chiama *pixel*. In Python rappresenteremo un'immagine tramite la lista delle righe della matrice dell'immagine ed ogni riga è la lista dei suoi pixels, ognuno dei quali è un colore.



## 12.1 RAPPRESENTAZIONE DEI COLORI

I colori sono rappresentati con combinazioni di tre colori primari, detti canali: rosso *red*, verde *green* e blu *blue*; *RGB* in breve. Per ogni canale si usa solitamente un intero tra 0 e 255 per indicare l'intensità di quel colore primario. Rappresentammo il colore di un pixel con la tupla  $(r, g, b)$ . Possiamo accedere al valore dei canali con l'unpacking.

```
bianco = (255,255,255)
nero = (0,0,0)
arancio = (255,128,0)
print(arancio)
# Out: (255, 128, 0)
r, g, b = arancio
print(r, g, b)
# Out: 255 128 0
```

Facciamo alcuni esempi di iterazione su colori che useremo successivamente.

```
colori = [(255,0,0),(0,255,0),(0,0,255)]

# colore è una tupla
for colore in colori:
    print(colore)
# Out: (255, 0, 0)
# Out: (0, 255, 0)
# Out: (0, 0, 255)

# unpacking delle tuple esplicito
for r, g, b in colori:
    r, g, b = colore
    print(r, g, b)
# Out: 0 0 255
# Out: 0 0 255
# Out: 0 0 255
```

```
# unpacking delle tuple implicito  
for r, g, b in colori:  
    print(r, g, b)  
# Out: 255 0 0  
# Out: 0 255 0  
# Out: 0 0 255
```

## 12.2 RAPPRESENTAZIONE DI IMMAGINI

Possiamo rappresentare immagini tramite una matrice di colori, utilizzando una lista di liste. La lista “esterna” è la lista delle righe dell’immagine. Ogni lista “interna” contiene i valori della corrispondente riga dell’immagine. Ad esempio una piccolissima immagine 2x2 si può rappresentare con

```
img = [  
    [ (255,0,0), ( 0,255,0) ],  
    [ (0,0,255), (255,128,0) ]  
]  
print(img)  
# Out: [[(255, 0, 0), (0, 255, 0)], [(0, 0, 255), (255, 128, 0)]]
```

L’accesso agli elementi avviene accedendo prima alla riga e poi alla colonna.

```
# prima riga (riga 0)  
print(img[0])  
# Out: [(255, 0, 0), (0, 255, 0)]  
# seconda riga (riga 1)  
print(img[1])  
# Out: [(0, 0, 255), (255, 128, 0)]  
  
# secondo elemento, prima riga (riga 0, colonna 1)  
print(img[0][1])  
# Out: (0, 255, 0)  
# primo elemento, seconda riga (riga 1, colonna 0)  
print(img[1][0])  
# Out: (0, 0, 255)
```

Le immagini rappresentate in questo modo hanno come altezza il numero di righe `len(img)` e come larghezza il numero delle colonne, che possiamo calcolare come la lunghezza della prima riga `len(img[0])`.

```
def width(img):  
    '''Ritorna la larghezza dell'immagine img.'''
```

```
    return len(img[0])

def height(img):
    '''Ritorna l'altezza dell'immagine img.'''
    return len(img)

w, h = width(img), height(img)
print(w, h)
# Out: 2 2
```

Possiamo iterare sui pixel dell'immagine iterando prima sulle righe e poi sulle colonne.

```
# iteriamo sulle righe
for riga in img:
    # iteriamo sulle colonne
    for colore in riga:
        print(colore)
# Out: (255, 0, 0)
# Out: (0, 255, 0)
# Out: (0, 0, 255)
# Out: (255, 128, 0)
```

## 12.3 SALVATAGGIO DI IMMAGINI

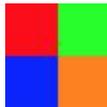
Da ora in avanti per vedere i risultati, salveremo le nostre immagini su disco in formato binario PNG utilizzando il modulo `png.py` scaricabile dalla libreria **PyPNG**. Per salvare un'immagine, chiamiamo la funzione `from_array()` per creare un'immagine PNG e la salviamo con il suo metodo `save()`.

```
import png

def save(filename, img):
    '''Salva un'immagine in formato PNG.'''
    pyimg = png.from_array(img, 'RGB')
    pyimg.save(filename)

save('img_small.png', img)
```

Questo salva nel file `img_small.png` la nostra piccolissima immagine, che è mostrata di seguito ingrandita.



## 12.4 CREAZIONE DI IMMAGINI

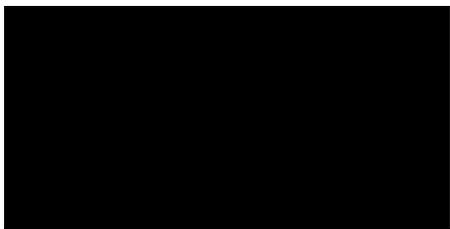
Per creare immagini più grandi, definiamo una funzione `create()` che crea un'immagine di larghezza `iw` e altezza `ih` con tutti i pixel settati al colore `c`. Per l'ultimo parametro utilizziamo un parametro opzionale, dato che useremo lo sfondo nero nella maggior parte degli esempi.

```
def create(iw, ih, c=(0,0,0)):
    '''Crea e ritorna un'immagine di larghezza iw,
    altezza ih e riempita con il colore c'''

    # L'immagine inizialmente vuota
    img = []
    # Per ogni riga,
    for _ in range(ih):
        # inizializza la riga vuota
        row = []
        # e per ogni pixel della riga,
        for _ in range(iw):
            # aggiunge un pixel di colore c.
            row.append(c)
        # Aggiunge la riga all'immagine
        img.append(row)
    return img
```

Ad esempio, la chiamata `create(256, 256)`, produce un'immagine nera di 256x256. Se stiamo usando la shell di IPython possiamo visualizzare *inline*, cioè direttamente sulla shell, le immagini che creiamo. Se eseguiamo il codice da terminale, salviamo l'immagine su disco per vederla.

```
img = create(256, 128)
save('img_create.png', img)
```



## 12.5 ACCESSO AI PIXEL

Possiamo creare immagini con `create()` e poi alterarne i pixels con varie funzioni, creando immagini più complesse dalla combinazione di funzioni semplici.

Iniziamo con una funzione `draw_quad_simple()` che disegna sull'immagine un rettangolo di colore dato specificandone l'angolo superiore sinistro e le dimensioni.

```
def draw_quad_simple(img, x, y, w, h, c):
    '''Disegna su img un rettangolo con lo spigolo in
    alto a sinistra in (x, y), larghezza w, altezza h
    e di colore c. Va in errore se il rettangolo
    fuoriesce dall'immagine.'''

    # Per ogni riga j del rettangolo,
    for j in range(y, y+h):
        # per ogni colonna i della riga j,
        for i in range(x, x+w):
            # imposta il colore del pixel a c
            img[j][i] = c

img = create(256,128)
draw_quad_simple(img, 16, 16, 224, 96, arancio)
save('img_quad_simple.png', img)
```



Nella funzione `draw_quad_simple()` si assume che l'utente specifichi un rettangolo che non esce dall'immagine. La funzione genera un errore se accede al di fuori del rettangolo dell'immagine.

```
draw_quad_simple(img, 16, 16, 512, 512, arancio)
# Error: Traceback (most recent call last):
# Error:   File "<input>", line 1, in <module>
# Error:   File "<input>", line 12, in draw_quad_simple
# Error: IndexError: list assignment index out of range
```

Per essere modificare la funzione precedente mettendo un test per controllare se le coordinate siano dentro l'immagine. Per fare ciò, ad ogni accesso confrontiamo gli indici di riga e colonna con le dimensioni dell'immagine.

```
def inside(img, i, j):
    '''Ritorna True se il pixel (i, j) è dentro
    l'immagine img, False altrimenti'''
    iw, ih = width(img), height(img)
    return 0 <= i < iw and 0 <= j < ih

def draw_quad(img, x, y, w, h, c):
    '''Disegna su img un rettangolo con lo spigolo
    in alto a sinistra in (x, y), larghezza w,
    altezza h e di colore c.'''
    for j in range(y,y+h):
        for i in range(x,x+w):
            # Disegna il pixel solo se è dentro
            if inside(img,i,j):
                img[j][i] = c

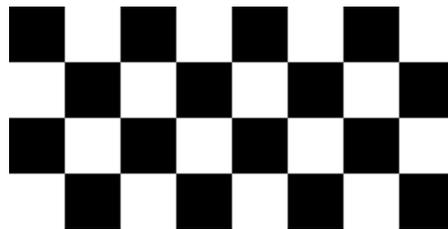
img = create(256, 128)
draw_quad(img, 16, 16, 512, 512, arancio)
save('img_quad.png',img)
```



Definiamo ora una funzione `draw_checkers()` che disegna una scacchiera di quadrati di lato dato colorati in modo alternato tra due colori. Per farlo,

calcoliamo gli indici di ogni quadrato dividendo le coordinate del pixel per la dimensione del quadrato utilizzando la divisione intera. Fatto questo, possiamo selezionare il colore facendo la somma degli indici modulo 2 alternando così i due colori sia per righe che per colonne.

```
def draw_checkers(img, s, c0, c1):  
    '''Disegna su img una scacchiera di quadratini,  
    ognuno di lato s, coi colori c0 e c1'''  
    # Per ogni indice di riga,  
    for jj in range(height(img)//s):  
        # per ogni indice di colonna  
        for ii in range(width(img)//s):  
            # seleziona il colore  
            if (ii + jj) % 2: c = c1  
            else: c = c0  
            # e disegna il quadratino  
            draw_quad(img, ii*s, jj*s, s, s, c)  
  
img = create(256, 128)  
draw_checkers(img, 32, (0,0,0), (255,255,255))  
save('img_checkers.png',img)
```

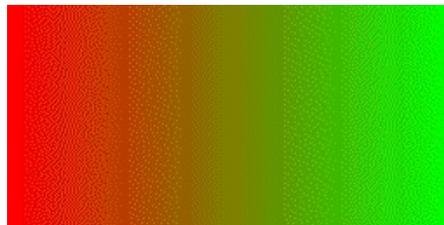


## 12.6 OPERAZIONI SUI COLORI

Oltre a disegnare sull'immagine possiamo operare direttamente sui colori. Ad esempio possiamo scrivere la funzione `draw_gradienth()` che disegna sull'immagine un gradiente di colore orizzontale interpolando due colori dati.

```
def draw_gradienth(img, c0, c1):
    '''Disegna su img un gradiente di colore da
    sinistra a destra, dal colore c0 al colore c1'''
    r0, g0, b0 = c0
    r1, g1, b1 = c1
    for j in range(height(img)):
        for i in range(width(img)):
            # float da 0 a 1
            u = i / width(img)
            # Interpolazione i canali
            r = round(r0 * (1-u) + r1 * u)
            g = round(g0 * (1-u) + g1 * u)
            b = round(b0 * (1-u) + b1 * u)
            img[j][i] = (r,g,b)

img = create(256, 128)
draw_gradienth(img, (255,0,0), (0,255,0))
save('img_gradienth.png',img)
```



In modo simmetrico possiamo definire la funzione che disegna un gradiente verticale.

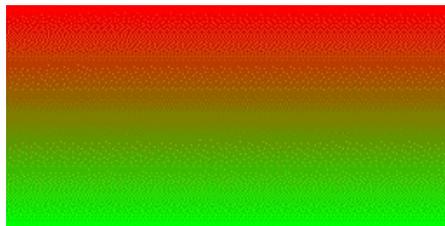
```
def draw_gradientv(img, c0, c1):
```

```

'''Disegna su img un gradiente di colore dall'
alto in basso, dal colore c0 al colore c1'''
r0, g0, b0 = c0
r1, g1, b1 = c1
for j in range(height(img)):
    for i in range(width(img)):
        v = j / height(img)
        r = round(r0 * (1-v) + r1 * v)
        g = round(g0 * (1-v) + g1 * v)
        b = round(b0 * (1-v) + b1 * v)
        img[j][i] = (r,g,b)

img = create(256, 128)
draw_gradientv(img, (255,0,0), (0,255,0))
save('img_gradientv.png',img)

```



Possiamo infine creare gradienti in tutte e due le direzioni contemporaneamente, interpolando i colori sui due assi. Questo combina i due esempi precedenti.

```

def draw_gradient_quad(img, c00, c01, c10, c11):
    '''Disegna un gradiente di colore combinato
    orizzontale e verticale con c00 in alto a
    sinistra, c01 in basso a sinistra, c10 in
    alto a destra e c11 in basso a destra'''
    for j in range(height(img)):
        for i in range(width(img)):
            u = i / width(img)
            v = j / height(img)
            c = [0,0,0]
            for k in range(3):
                c[k] = round(c00[k]*(1-u)*(1-v) +
                             c01[k]*(1-u)*v +
                             c10[k]*u*(1-v) +
                             c11[k]*u*v)

```

```
img[j][i] = tuple(c)

img = create(256, 128)
draw_gradient_quad(img, (255,0,0), (0,255,0),
                      (0,0,255), (255,255,255))
save('img_gradientq.png',img)
```



## 12.7 CARICAMENTO DI IMMAGINI

Per caricare le immagini, useremo ancora il modulo `png.py` nel definire una funzione `load(filename)` che carica un'immagine PNG da `filename`. In questo caso la funzione è più complessa e ne riportiamo il codice commentato qui sotto.

```
def load(filename):
    '''Carica l'immagine in formato PNG dal file
    filename, la converte nel formato a matrice
    di tuple e la ritorna'''
    with open(filename,'rb') as f:
        # legge l'immagine come RGB a 256 valori
        r = png.Reader(file=f)
        iw, ih, png_img, _ = r.asRGB8()
        # converte in lista di liste di tuple
        img = []
        for png_row in png_img:
            row = []
            # l'immagine PNG ha i colori in
            # un'unico array quindi li leggiamo
            # tre alla volta in una tupla
            for i in range(0, len(png_row), 3):
                row.append( ( png_row[i+0],
                             png_row[i+1],
                             png_row[i+2] ) )
            img.append( row )
    return img
```

Per testare la funzione, leggiamo e scriviamo lo stesso file. Nel resto di questo capitolo si possono usare immagini arbitrarie, ma noi consigliamo l'uso di foto.

```
img = load('photo.png')
save('img_photo.png',img)
```



## 12.8 COPIE E CORNICI

Se vogliamo aggiungere una cornice di un certo colore ad un'immagine possiamo farlo creando una nuova immagine riempita con il colore della cornice, grande tanto da contenere l'immagine originale più la cornice, e poi copiamo l'immagine originale al centro della nuova immagine. Definiremo la funzione di copia in modo generico perché possa poi essere utilizzata per fare altre manipolazioni.

```
def copy(dst, src, dx, dy, sx, sy, w, h):
    '''Copia la porzione rettangolare dell'immagine
    src con spigolo in alto a sinistra in (sx, sy)
    e dimensioni w, h sull'immagine dst a partire
    da (dx, dy)'''
    for j in range(h):
        for i in range(w):
            di, dj = i+dx, j+dy
            si, sj = i+sx, j+sy
            if (inside(dst, di, dj) and
                inside(src, si, sj)):
                dst[dj][di] = src[sj][si]

def border(img, s, c):
    '''Ritorna una nuova immagine che è l'immagine
    img contornata da una cornice di spessore s
    e colore c'''
    w, h = width(img), height(img)
    ret = create(w+s*2, h+s*2, c)
    copy(ret, img, s, s, 0, 0, w, h)
    return ret

save('img_border.png', border(img, 8, (0,0,0)))
```



## 12.9 ROTAZIONI

Possiamo ruotare un'immagine intorno all'asse verticale o a quello orizzontale. Per ruotarla intorno all'asse verticale, basta scambiare tra loro i pixels di ogni riga che si trovano alla stessa distanza dal centro dell'immagine.

```
def fliph(img):  
    '''Ritorna una nuova immagine che e' l'immagine  
    img ruotata intorno al suo asse verticale, cioè  
    i pixels sono scambiati orizzontalmente'''  
    w, h = width(img), height(img)  
    ret = create(w, h)  
    for j in range(h):  
        for i in range(w):  
            ret[j][i] = img[j][w - 1 - i]  
    return ret  
  
save('img_fliph.png', fliph(img))
```



In modo simmetrico possono implementare la rotazione intorno all'asse orizzontale.

```
def flipv(img):  
    '''Ritorna una nuova immagine che è l'immagine  
    img ruotata intorno al suo asse orizzontale,
```

```

cioè i pixels sono scambiati verticalmente'''
w, h = width(img), height(img)
ret = create(w, h)
for j in range(h):
    for i in range(w):
        ret[j][i] = img[h - 1 - j][i]
return ret

save('img_flipv.png', flipv(img))

```



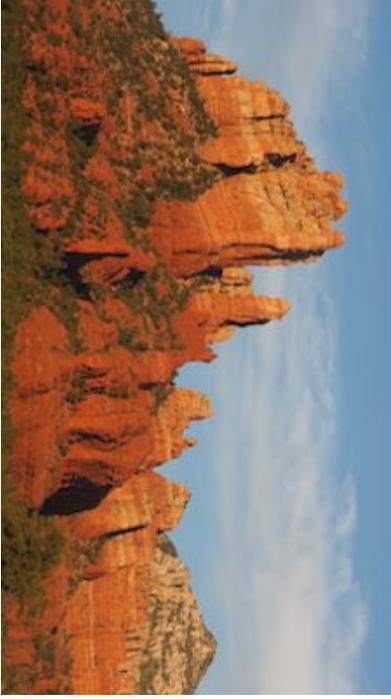
Infine, consideriamo la rotazione attorno all'angolo inferiore sinistro. Questa equivale ad invertire le righe con le colonne. In questo caso, l'immagine creata avrà altezza e larghezza invertite rispetto a quella di input.

```

def rotate(img):
    '''Ritorna una nuova immagine che è l'immagine
    img ruotata intorno all'angolo inferiore
    sinistro, equivalente a scambiare le righe
    con le colonne'''
    w, h = width(img), height(img)
    # altezza e larghezza sono invertite
    ret = create(h, w)
    for j in range(h):
        for i in range(w):
            ret[i][j] = img[h-1-j][i]
    return ret

save('img_rotate.png', rotate(img))

```



## 12.10 MODIFICA DEI COLORI

Possiamo modificare i colori di un'immagine per creare effetti interessanti o semplicemente per migliorare l'aspetto dell'immagine. Una modifica molto semplice è l'inversione che corrisponde a creare il "negativo" dell'immagine, prendendo per ogni canale il valore 255 e sottraendo il valore del canale. Questo ad esempio trasforma il bianco in nero e viceversa.

```
def invert(img):  
    '''Ritorna una nuova immagine che è l'immagine  
    img con colori invertiti'''  
    w, h = width(img), height(img)  
    ret = create(w, h, (0,0,0))  
    for j in range(h):  
        for i in range(w):  
            r, g, b = img[j][i]  
            ret[j][i] = (255 - r, 255 - g, 255 - b)  
    return ret  
  
save('img_invert.png',invert(img))
```



Possiamo scrivere molti altri esempi di manipolazione del colore e tutti avrebbero la stessa forma, cioè l'applicazione ad ogni pixel di una trasformazione dei canali. Per sfruttare questa similarità introduciamo una funzione `filter()` che altera i colori di una immagine applicando una funzione `func` presa come input. Con questa funzione possiamo

implementare lo stesso effetto della funzione `invert()`.

```
def filter(img,func):  
    '''Ritorna una nuova immagine che è l'immagine  
    img con colori filtrati da func'''  
    w, h = width(img), height(img)  
    ret = create(w, h, (0,0,0))  
    for j in range(h):  
        for i in range(w):  
            r, g, b = img[j][i]  
            ret[j][i] = func(r, g, b)  
    return ret  
  
def invertf(r,g,b):  
    return 255 - r, 255 - g, 255 - b  
  
save('img_invertf.png',filter(img,invertf))
```

Adesso che abbiamo `filter()` possiamo applicare qualunque trasformazione di colori, come ad esempio la conversione in bianco e nero, che setta ogni canale alla media dei loro valori, o applicare del contrasto all'immagine, scalando i canali rispetto al valore medio 128.

```
def grayf(r,g,b):  
    gray = (r + g + b) // 3  
    return gray, gray, gray  
  
save('img_grayf.png',filter(img,grayf))
```



```
def contrastf(r,g,b):  
    return ( max(0,min(255, (r - 128) * 2 + 128)),  
            max(0,min(255, (g - 128) * 2 + 128)),  
            max(0,min(255, (b - 128) * 2 + 128)))  
  
save('img_contrastf.png',filter(img,contrastf))
```



## 12.11 MOSAICI

Possiamo creare un mosaico da un'immagine dividendola in quadrati e riempiendo ogni quadrato con un solo colore che dipende dai colori dei pixel dell'immagine originale contenuti nel quadrato stesso. Nella prima versione scegliamo di colorare ogni quadrato con il colore del suo pixel nell'angolo in alto a sinistra.

```
def mosaic_nearest(img, s):  
    '''Ritorna una nuova immagine ottenuta dividendo  
    l'immagine img in quadrati di lato s e riempiendo  
    ogni quadrato con il colore del suo angolo in  
    alto a sinistra'''  
    w, h = width(img), height(img)  
    ret = create(w, h)  
    # itera sui possibili quadrati  
    for jj in range(h//s):  
        for ii in range(w//s):  
            # colore dell'angolo in alto-sinistra  
            c = img[jj*s][ii*s]  
            draw_quad(ret, ii*s, jj*s, s, s, c)  
    return ret  
  
save('img_mosaicn.png', mosaic_nearest(img, 16))
```



Come si può notare l'immagine è poco riconoscibile. Possiamo invece

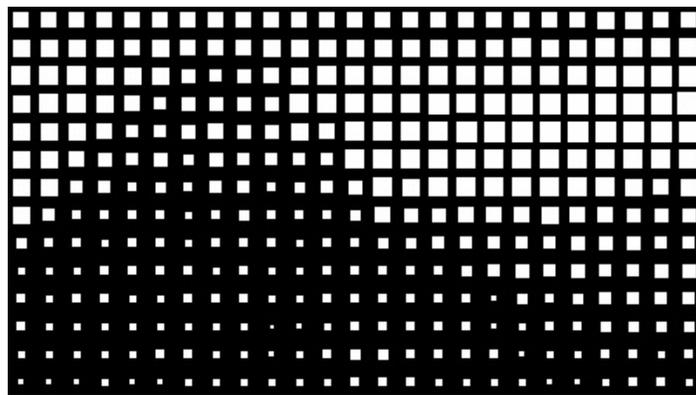
scegliere il colore di ogni quadrato facendo la media dei colori dei pixel del quadrato.

```
def average(img, i, j, w, h):  
    '''Calcola la media dei valori dell'area  
    [i,w-1]x[j,h-1].'''  
    c = [0,0,0]  
    for jj in range(j,j+h):  
        for ii in range(i,i+w):  
            for k in range(3):  
                c[k] += img[jj][ii][k]  
    for k in range(3):  
        c[k] //= w*h  
    return tuple(c)  
  
def mosaic_average(img, s):  
    '''Ritorna una nuova immagine ottenuta dividendo  
    l'immagine img in quadrati di lato s e riempiendo  
    ogni quadratino con la media dei suoi colori.'''  
    w, h = width(img), height(img)  
    ret = create(w, h)  
    # itera sui possibili quadrati  
    for jj in range(h//s):  
        for ii in range(w//s):  
            # colore medio dell'immagine  
            c = average(img,ii*s,jj*s,s,s)  
            draw_quad(ret, ii*s, jj*s, s, s, c)  
    return ret  
  
save('img_mosaica.png',mosaic_average(img,16))
```



Un modo piuttosto diverso di creare un mosaico è di disegnare dentro ogni quadrato un quadrato bianco centrale, su sfondo nero, di lato proporzionale alla luminosità media della parte di immagine considerata.

```
def mosaic_size(img, s):  
    '''Ritorna una nuova immagine ottenuta dividendo  
    l'immagine img in quadratini di lato s e  
    disegnando all'interno di ognuno di essi,  
    su sfondo nero, un quadratino centrale bianco di  
    lato proporzionale alla luminosità media del  
    corrispondente quadratino'''  
    w, h = width(img), height(img)  
    ret = create(w, h)  
    # itera sui possibili quadrati  
    for jj in range(h//s):  
        for ii in range(w//s):  
            # colore medio dell'immagine  
            c = average(img, ii*s, jj*s, s, s)  
            # lato del quadratino bianco  
            r = round(s*(c[0]+c[1]+c[2])/(3*255))  
            draw_quad(ret, ii*s+(s-r)//2,  
                    jj*s+(s-r)//2, r, r, (255,255,255))  
    return ret  
  
save('img_mosaics.png', mosaic_size(img, 16))
```



## 12.12 SPOSTAMENTO DI PIXELS

Possiamo modificare il colore di un pixel copiando il colore di un altro pixel scelto in modo casuale ma a distanza massima prefissata. Scegliere valori in modo casuale è molto per ottenere variazioni che non si ripetono. La libreria standard di Python ha il modulo `random` che contiene funzioni per la generazione di numeri casuali. In particolare, le funzioni `randint(a, b)` e `uniform(a, b)` generano, ad ogni chiamata, rispettivamente un intero e un numero reale random compreso tra `a` e `b`. Per riprodurre la stessa sequenza di numeri casuali ad ogni chiamata di una funzione si può usare la funzione `seed(value)` con lo stesso seme `value`.

```
import random

def scramble(img, d, s):
    '''Ritorna una nuova immagine ottenuta colorando
    ogni pixel (i, j) con il colore di un pixel
    scelto a caso nel quadratino centrato in (i, j)
    di lato 2*d + 1'''
    # settiamo il seed per generare la stessa
    # sequenza di numeri casuali
    random.seed(s)
    w, h = width(img), height(img)
    ret = create(w, h)
    for j in range(h):
        for i in range(w):
            # sceglie a caso un pixel nel quadrato
            ri = i + random.randint(-d,d)
            rj = j + random.randint(-d,d)
            # evitando che si esca dall'immagine
            ri = max(0, min(w-1, ri))
            rj = max(0, min(h-1, rj))
            ret[j][i] = img[rj][ri]
    return ret

save('img_scramble.png',scramble(img,16,0))
```

Infine, per creare un effetto lente, basta spostare i pixel lungo la linea che collega il centro della lente e la posizione del pixel stesso. Lo spostamento effettuale può essere controllata semplicemente elevando a potenza la distanza pixel-centro, dove la potenza scelta definisce le caratteristiche della lente.

```
import math

def lens(img, x, y, r, p):
    '''Ritorna una nuova immagine ottenuta dall'
    immagine img applicando una lente di raggio r,
    centrata in (x, y) e di power p. Se p = 1.0 la
    lente non distorce, se p > 1.0 la lente
    ingrandisce e se p < 1.0 riduce.'''
    w, h = width(img), height(img)
    ret = create(w, h)
    for j in range(h):
        for i in range(w):
            di, dj = i - x, j - y
            # distanza al quadrato da (x, y)
            d2 = di*di + dj*dj
            # se è nel raggio della lente
            if d2 < r*r:
                rr = math.sqrt(d2) / r
                if rr > 0:
                    ratio = (rr ** p) / rr
                else:
                    ratio = 1.0
                li = int((i-x)*ratio+x)
                lj = int((j-y)*ratio+y)
                if inside(img, li, lj):
                    ret[j][i] = img[lj][li]
                else:
                    ret[j][i] = (0,0,0)
            else:
                ret[j][i] = img[j][i]
    return ret

save('img_lens.png', lens(img, 128, 128, 100, 0.5))
save('img_lensb.png', lens(img, 128, 128, 100, 2.0))
```



## 13 TIPI DEFINITI DALL'UTENTE

Questo capitolo introduce i concetti necessari per definire nuovi tipi in Python e ne dimostra l'utilità riscrivendo alcuni esempi già visti di elaborazione di immagini usando i nuovi tipi introdotti.

## 13.1 CLASSI

Python, al pari di altri linguaggi orientati agli oggetti, permette di introdurre nuovi tipi tramite il concetto di *classe*. Una classe è la definizione di un tipo i cui valori sono oggetti che hanno uno stato, memorizzato in variabili specifiche per ogni oggetto, e delle operazioni, definite da metodi. Le variabili e i metodi definiti in una classe sono spesso chiamati *attributi*. Tutti i tipi di Python, inclusi ad esempio `int` e `list`, sono definiti tramite classi e tutte le operazioni che possono essere eseguite su di essi sono definite come metodi nella loro classe.

Le classi non sono necessarie per risolvere nuovi problemi. Infatti quello che si può elaborare con le classi si può anche elaborare senza di esse. Tuttavia l'uso delle classi spesso rende il codice più leggibile per due motivi. Per primo, le classi associano, in modo esplicito, il tipo di un oggetto con le operazioni definite su quel tipo di dati, attraverso la definizione di metodi. Inoltre, le classi permettono di nascondere i dettagli implementativi che non servono ai fini dell'utilizzo del nuovo tipo. Al contrario, usare solamente tipi di base e funzioni porta a dover esplicitare più dettagli implementativi che poi devono essere ricordati durante la programmazione. Ad esempio, nella manipolazione dei colori abbiamo sempre esplicitato il fatto che un colore è rappresentato tramite una tupla invece di esprimere un colore con oggetto di un tipo specifico, ad esempio `Color`. Stessa cosa per le immagini che abbiamo manipolato esplicitamente come liste di liste, invece di usare un oggetto più specifico di tipo `Image`.

## 13.2 COSTRUTTORE

Per definire una classe in Python si usa la parola chiave `class` seguita dal nome che si vuole dare al nuovo tipo. Per convenzione i tipi hanno nomi che iniziano con una lettera maiuscola. Il nome è seguito dall'elenco dei metodi, ognuno dei quali prende come primo argomento la variabile `self` che indica l'oggetto su cui si sta agendo. Le variabili che definiscono lo stato di ogni oggetto sono definite attraverso un metodo speciale, chiamato *costruttore*, ed indicato col nome `__init__()`.

```
class NomeTipo:
    def __init__(self, parametri):
        self.nome_variabile = valore
        ...
    definizione dei metodi ...
```

Iniziamo a fare un esempio con la classe `Color` che rappresenta colori. Ogni *oggetto* di tipo `Color` rappresenta un colore specifico, ad esempio nero o bianco, determinato assegnando opportuni valori a tre variabili che ne definiscono lo stato. Possiamo definire una classe `Color` minimale specificando solo il costruttore.

```
class Color:
    def __init__(self, r, g, b):
        self.r = r
        self.g = g
        self.b = b
```

Il parametro `self` è speciale e deve sempre essere il primo parametro di un qualsiasi metodo della classe. Python assegna automaticamente al parametro `self` l'oggetto relativamente al quale il metodo è stato chiamato, quindi non deve essere specificato nella chiamata del metodo. Nel caso del costruttore, il valore di `self` è l'oggetto che si sta costruendo. La sintassi del `.`, come già sappiamo, permette di accedere agli attributi di un oggetto che

possono essere metodi o variabili. Nel nostro caso `self.r`, `self.g` e `self.b` sono variabili che vengono definite per gli oggetti di tipo `Color` e che contengono valori numerici.

## 13.3 OGGETTI

Gli oggetti di una classe si creano chiamando il costruttore che è invocato col nome della classe usato come se fosse il nome di una funzione e con gli argomenti che vogliamo passare al costruttore.

```
# crea un oggetto e poi chiama
# il costruttore NomeTipo.__init__
oggetto = NomeTipo(argomenti)
```

L'oggetto creato è del tipo della classe e possiamo accedere alle sue variabili usando la sintassi del punto.

```
# Creazione di un oggetto di tipo Color
c1 = Color(255,0,0)
print(type(c1))
# Out: <class '__console__.Color'>

# Valore dell'attributo r dell'oggetto creato
print(c1.r)
# Out: 255
print(c1.g)
# Out: 0
print(c1.b)
# Out: 0
```

Creando due oggetti dello stesso tipo, i valori delle rispettive variabili sono diverse.

```
# Un altro oggetto di tipo Color
c2 = Color(0,255,0)
print(type(c2))
# Out: <class '__console__.Color'>

# Il valore dell'attributo r è diverso
print(c2.r)
```

```
# Out: 0
print(c2.g)
# Out: 255
print(c2.b)
# Out: 0

# Infatti i due oggetti hanno identità differenti
print(id(c1))
# Out: 4427973240
print(id(c2))
# Out: 4427973520
```

Cercando di accedere ad attributi non esistenti, otteniamo un errore.

```
print(c1.z)
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: AttributeError: 'Color' object has no attribute 'z'
```

Gli oggetti in Python sono *mutabili*, cioè possiamo accedere ad una qualsiasi degli variabili e cambiarne il valore. Ovviamente questo modifica solamente il valore dell'attributo di uno specifico oggetto, non di altri oggetti.

```
# Modifica l'attributo g dell'oggetto in c1
c1.g = 128
print(c1.g)
# Out: 128

# Il corrispondente attributo di c2 non è cambiato
print(c2.g)
# Out: 255
```

## 13.4 METODI

Le operazioni su oggetti sono definite tramite *metodi* e agiscono, tipicamente, sulle variabili dell'oggetto sul quale sono chiamati. Abbiamo già usato metodi predefiniti su stringhe e liste, ad esempio `append()` e `split()`. La sintassi per definire un metodo è simile a quella per il costruttore. Il primo parametro deve sempre essere `self` che contiene l'oggetto su cui agisce il metodo.

```
class NomeTipo:
    def nome_metodo(self, parametri):
        istruzioni
```

Aggiungiamo un metodo alla classe `Color` che crea un nuovo oggetto con colore inverso e lo ritorna.

```
class Color:
    def __init__(self, r, g, b):
        self.r, self.g, self.b = r, g, b
    def inverse(self):
        return Color(255 - self.r,
                    255 - self.g, 255 - self.b)
```

Possiamo chiamare un metodo usando la solita sintassi già presentata. Come per l'accesso alle variabili, un metodo può essere chiamato solamente in riferimento ad uno specifico oggetto.

```
c = Color(255,0,0)
print(c.r, c.g, c.b)
# Out: 255 0 0
ci = c.inverse()
print(ci.r, ci.g, ci.b)
# Out: 0 255 255
```

## 13.5 METODI SPECIALI

Il nostro tipo `Color` non è però equivalente ai tipi predefiniti. Ad esempio non possiamo stamparlo né applicargli operazioni aritmetiche.

```
# Creiamo due colori
c1 = Color(255,0,0)
c2 = Color(0,255,0)

# Proviamo a stamparli
print(c1)
# Out: <__console__.Color object at 0x107ed85f8>
print(c1.r, c1.g, c1.b)
# Out: 255 0 0

# Proviamo a sommarli
c3 = c1 + c2
# Error: Traceback (most recent call last):
# Error: File "<input>", line 1, in <module>
# Error: TypeError: unsupported operand type(s) for +: 'Color' and
'Color'
```

Oltre ai costruttori ci sono molti altri metodi speciali, con nomi che iniziano e finiscono con doppio underscore `__`, che permettono di dare ai tipi definiti da un programmatore un comportamento simile a quello dei tipi predefiniti. Ciò è molto utile per estendere il linguaggio e adattarlo a problemi specifici, ma mentendo una certa eleganza nella sintassi. Nel nostro caso, definiremo i metodi `__str__`, chiamato quando l'oggetto è convertito in stringa o stampato con `print`, e i metodi `__add__` e `__mul__` usati per addizione e moltiplicazione.

```
class Color:
    def __init__(self, r, g, b):
        self.r, self.g, self.b = r, g, b
    def inverse(self):
        return Color(255 - self.r,
```

```
        255 - self.g, 255 - self.b)
def __str__(self):
    return 'Color({}, {}, {})'.format(
        self.r, self.g, self.b)
def __add__(self, other):
    return Color(self.r+other.r,
        self.g+other.g, self.b+other.b)
def __mul__(self, f):
    return Color(self.r*f, self.g*f, self.b*f)
```

```
# Creiamo due colori
c1 = Color(255,0,0)
c2 = Color(0,255,0)

# Stampa di un colore
print(c1)
# Out: Color(255,0,0)

# Operazioni su colori
c3 = c1 + c2
print(c3)
# Out: Color(255,255,0)
c4 = c3*0.7
print(c4)
# Out: Color(178.5,178.5,0.0)
```

## 13.6 INCAPSULAMENTO

Le classi, se usate in modo adeguato, permettono di nascondere i dettagli con cui i metodi sono implementati. Questo è importante perché migliora la leggibilità del codice. Ad esempio, per usare `list.append()` è sufficiente sapere cosa fa il metodo, non come lo fa. Detto in un altro modo, non è importante sapere come il metodo è implementato. In Python si usa una convenzione per nascondere i dettagli implementativi: gli attributi i cui nomi iniziano con `_` sono da considerarsi nascosti, cioè non dovrebbero essere usati dall'esterno della classe.

Per illustrare questo concetto e vedere anche un esempio un po' più complesso, introduciamo la classe `Image` per rappresentare immagini. Nel farlo re-implementeremo come metodi alcune funzioni viste nel capitolo precedente. La classe `Image` rappresenterà i pixel tramite oggetti `Color` e il costruttore sarà analogo alla funzione `create()`. Aggiungiamo poi dei metodi per ritornare le dimensioni dell'immagine, per leggere e impostare i singoli pixel, e per leggere e salvare l'immagine su disco con `png.py`. Nel metodo `set_pixel()` i canali colore sono impostati individualmente perché non vogliamo sostituire l'oggetto `Color` con uno nuovo, ma solamente i valori dei suoi attributi `r`, `g` e `b`.

```
import png

class Image:
    def __init__(self, w, h):
        '''Crea un'immagine di dimensioni w x h
        riempita con colore nero'''
        # L'attributo _pixels deve rimanere nascosto
        self._pixels = []
        for j in range(h):
            row = []
            for i in range(w):
                row.append(Color(0,0,0))
            self._pixels.append(row)
    def width(self):
```

```

    '''Ritorna la larghezza dell'immagine'''
    return len(self._pixels[0])
def height(self):
    '''Ritorna l'altezza dell'immagine'''
    return len(self._pixels)
def set_pixel(self, i, j, color):
    '''Imposta il colore del pixel (i, j)'''
    if (0 <= i < self.width() and
        0 <= j < self.height()):
        self._pixels[j][i].r = color.r
        self._pixels[j][i].g = color.g
        self._pixels[j][i].b = color.b
def get_pixel(self, i, j):
    '''Ritorna l'oggetto Color del pixel (i,j)'''
    if (0 <= i < self.width() and
        0 <= j < self.height()):
        return self._pixels[j][i]
def load(self, filename):
    '''Carica l'immagine dal file filename'''
    with open(filename,'rb') as f:
        r = png.Reader(file=f)
        iw, ih, png_img, _ = r.asRGB8()
        img = []
        for png_row in png_img:
            row = []
            for i in range(0,len(png_row),3):
                row.append( Color(png_row[i+0],
                                png_row[i+1],png_row[i+2]) )
            img.append( row )
def save(self, filename):
    '''Salva l'immagine nel file filename'''
    pixels = []
    for j in range(self.height()):
        pixels.append([])
        for i in range(self.width()):
            c = self.get_pixel(i,j)
            pixels[-1] += [c.r,c.g,c.b]
    pyimg = png.from_array(pixels, 'RGB')
    pyimg.save(filename)
def draw_quad(self, x, y, w, h, c):
    '''Disegna sull'immagine un rettangolo con
    spigolo in (x,y), dimensioni wxh e
    colore c'''

```

```

    for j in range(y, y+h):
        for i in range(x, x+w):
            self.set_pixel(i,j,c)
def draw_gradienth(self, c0, c1):
    '''Disegna sull'immagine un gradiente
    orizzontale dal colore c0 al colore c1'''
    for j in range(self.height()):
        for i in range(self.width()):
            u = float(i) / float(self.width())
            self.set_pixel(i,j,c0*(1-u)+c1*u)
def __str__(self):
    return 'Image@{x{}}'.format(
        self.width(),self.height())

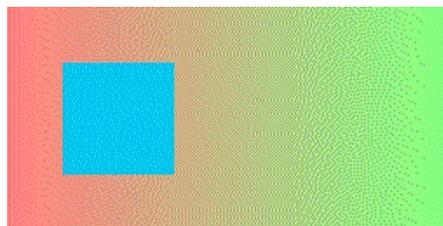
```

Da notare come il codice delle funzioni `draw_XXX()` è più leggibile, eliminando check espliciti a `inside()`, e molto meno pronò ad errori. Inoltre, potremmo cambiare la codifica dell'immagine alternando il costruttore e l'implementazione di alcuni metodi, ad esempio `set_pixel()` e `width()`, senza dover cambiare l'implementazione dei metodi `draw_XXX()`. Infine, le operazioni di somma e moltiplicazione per un fattore della classe `Color` permettono di definire in modo molto più semplice i metodi dei gradienti. Potremmo ovviamente implementare molti altri metodi sulla linea delle manipolazioni fatte in precedenza, ma lasciamo questo come esercizio al lettore.

```

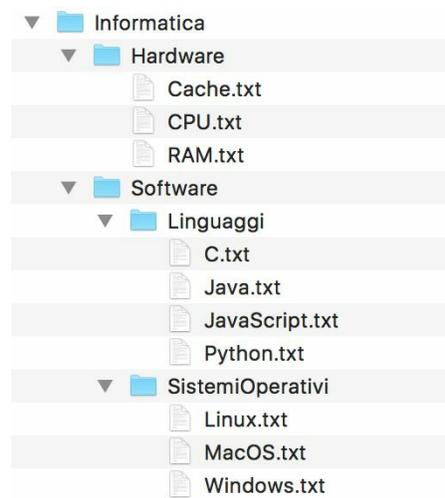
img = Image(256,128)
img.draw_gradienth(Color(255,128,128),
    Color(128,255,128))
img.draw_quad(32,32,64,64,Color(0,200,255))
img.save('img_draw.png')
print(img)
# Out: Image@256x128

```



## 14 ESPLORARE IL FILE SYSTEM

Questo capitolo introdurrà la *ricorsione*, una tecnica di programmazione che permette di elaborare strutture ad *albero*, come ad esempio la struttura di file e cartelle nei sistemi operativi. Il *file system*, cioè la collezione di file e cartelle su disco, è organizzato in modo gerarchico dove ogni cartella, o *directory*, contiene files ed altre directory, che a loro volta possono contenere altri file e directory. Gli esempi di questo capitolo si riferiscono ad una piccola porzione di file system rappresentata qui di seguito.



## 14.1 ESPLORE IL FILE SYSTEM

Prima di tutto scriviamo una funzione che stampa il contenuto di una directory. Ci avvarremo del modulo `os` di Python che offre funzioni per accedere a vari aspetti del sistema operativo. La funzione `os.listdir(dir)` ritorna una sequenza contenente i nomi dei file e directory contenute direttamente nella directory `dir`. La funzione `os.path.join(p1, p2, ...)` concatena, in modo intelligente, cioè secondo le regole del sistema operativo che si sta usando, due o più componenti di un percorso. Ad esempio stampiamo il contenuto della directory `Informatica`.

```
import os

def print_dir(dirpath):
    '''Stampa i percorsi di file e directory
    contenute nella directory dirpath'''
    for name in os.listdir(dirpath):
        # per evitare file nascosti
        if name.startswith('.'): continue
        print(os.path.join(dirpath, name))

print_dir('Informatica')
# Out: Informatica/Hardware
# Out: Informatica/Software
```

Se vogliamo stampare i contenuti delle directory contenute in `'Informatica'`, possiamo richiamare la funzione su ciascuna *subdirectory*.

```
print_dir('Informatica/Hardware')
# Out: Informatica/Hardware/Cache.txt
# Out: Informatica/Hardware/CPU.txt
# Out: Informatica/Hardware/RAM.txt

print_dir('Informatica/Software')
# Out: Informatica/Software/Linguaggi
# Out: Informatica/Software/SistemiOperativi
```

Se vogliamo stampare anche i contenuti delle subdirectory a qualsiasi livello di profondità, dovremmo richiamare la funzione su ogni subdirectory che incontriamo durante l'esplorazione. Il modo più naturale per farlo è di chiamare richiamare la stessa funzione su ogni subdirectory che incontriamo. Infatti, la funzione di esplorazione deve fare *ricorsivamente* la stessa cosa su ogni directory. Per riconoscere se un certo percorso si riferisce a una directory possiamo usare la funzione `os.path.isdir(path)`.

```
def print_dir_tree(dirpath):
    '''Stampa i percorsi di tutti i file e directory
    contenuti, a qualsiasi livello, nella directory
    dirpath'''
    for name in os.listdir(dirpath):
        # per evitare file nascosti
        if name.startswith('.'): continue
        pathname = os.path.join(dirpath, name)
        print(pathname)
        # se e' una directory
        if os.path.isdir(pathname):
            # chiama ricorsivamente la funzione
            print_dir_tree(pathname)

print_dir_tree('Informatica')
# Out: Informatica/Hardware
# Out: Informatica/Hardware/Cache.txt
# Out: Informatica/Hardware/CPU.txt
# Out: Informatica/Hardware/RAM.txt
# Out: Informatica/Software
# Out: Informatica/Software/Linguaggi
# Out: Informatica/Software/Linguaggi/C.txt
# Out: Informatica/Software/Linguaggi/Java.txt
# Out: Informatica/Software/Linguaggi/JavaScript.txt
# Out: Informatica/Software/Linguaggi/Python.txt
# Out: Informatica/Software/SistemiOperativi
# Out: Informatica/Software/SistemiOperativi/Linux.txt
# Out: Informatica/Software/SistemiOperativi/MacOS.txt
# Out: Informatica/Software/SistemiOperativi/Windows.txt
```

Questo semplice esempio illustra una tecnica fondamentale della programmazione detta *ricorsione*. L'idea è che una funzione può richiamare

se stessa su input diversi. La ricorsione risulta utile quando la risoluzione di un problema richiede la risoluzione di sottoproblemi dello stesso tipo, per cui il metodo risolutivo può essere applicato anche ad essi ricorsivamente. Nel nostro esempio, l'esplorazione di una directory richiede l'esplorazione delle sue subdirectories.

## 14.2 RICORSIONE

Vediamo un altro semplice esempio per cui la ricorsione è utile. Data una sequenza `seq` (lista, stringa, ecc.) vogliamo costruire una lista di tutte le permutazioni degli elementi di `seq`. Ad esempio, se `seq = [1,2,3]` la funzione deve ritornare la lista

```
[[1,2,3], [1,3,2], [2,1,3], [2,3,1], [3,1,2], [3,2,1]]
```

Un modo per generare le permutazioni di una sequenza è di generare, per ogni elemento, tutte le permutazioni in cui esso è in testa. Per falso, basta generare le permutazioni dei rimanenti elementi e poi aggiungere l'elemento scelto in testa. Così abbiamo ricondotto la generazione delle permutazioni di una sequenza di lunghezza  $n$  alla generazione delle permutazioni di sequenze di lunghezza  $n-1$ .

```
def permute(seq):
    '''Ritorna la lista di tutte le permutazioni
    della sequenza seq'''
    if len(seq) <= 1:
        perms = [seq]
    else:
        perms = []
        for i in range(len(seq)):
            # genera ricorsivamente le permutazioni
            # degli elementi escluso l'i-esimo
            sub = permute(seq[:i]+seq[i+1:])
            for p in sub: # mette in testa l'i-esimo elemento
                perms.append(seq[i:i+1]+p)
        return perms

print(permute([1,2,3]))
# Out: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3,
2, 1]]
```

Per visualizzare meglio l'albero delle chiamate ricorsive, aggiungiamo la stampa di qualche messaggio alla funzione.

```
def permute_print(seq):
    '''Ritorna la lista di tutte le permutazioni
    della sequenza seq'''
    print(' '*3-len(seq), 'chiamata', seq)
    if len(seq) <= 1:
        perms = [seq]
    else:
        perms = []
        for i in range(len(seq)):
            sub = permute_print(seq[:i]+seq[i+1:])
            for p in sub:
                perms.append(seq[i:i+1]+p)
    print(' '*3-len(seq), 'ritorna', perms)
    return perms
```

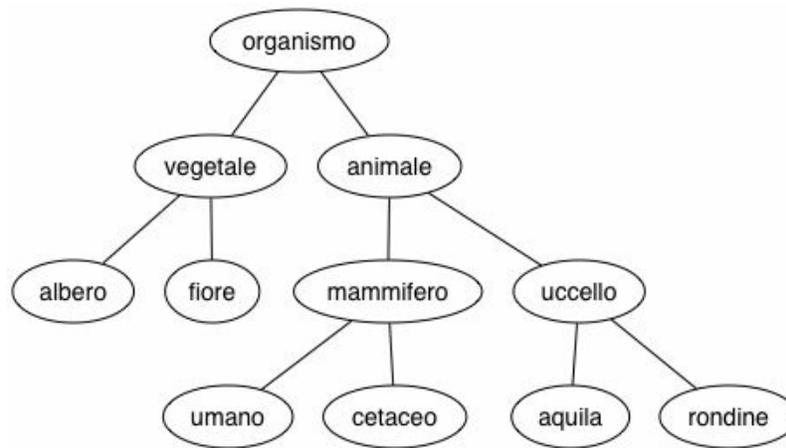
```
permute_print([1,2,3])
# Out: chiamata [1, 2, 3]
# Out: chiamata [2, 3]
# Out: chiamata [3]
# Out: ritorna [[3]]
# Out: chiamata [2]
# Out: ritorna [[2]]
# Out: ritorna [[2, 3], [3, 2]]
# Out: chiamata [1, 3]
# Out: chiamata [3]
# Out: ritorna [[3]]
# Out: chiamata [1]
# Out: ritorna [[1]]
# Out: ritorna [[1, 3], [3, 1]]
# Out: chiamata [1, 2]
# Out: chiamata [2]
# Out: ritorna [[2]]
# Out: chiamata [1]
# Out: ritorna [[1]]
# Out: ritorna [[1, 2], [2, 1]]
# Out: ritorna [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
# Out: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

---

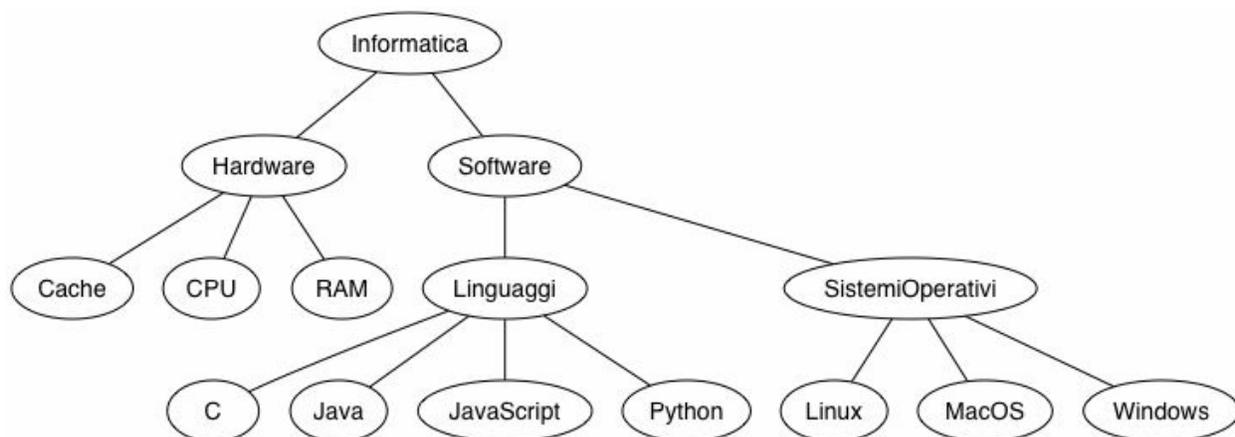
In questo secondo esempio la ricorsione non esplora un albero esplicito, come quello dei file e delle sottodirectory, ma visita un albero implicito prodotto dalla nidificazione delle chiamate ricorsive.

## 14.3 ALBERI

I semplici esempi che abbiamo visto illustrano il legame tra la ricorsione e gli alberi. Gli alberi si presentano in tantissime situazioni. Ad esempio, sono frequentemente usati per rappresentare classificazioni o tassonomie.



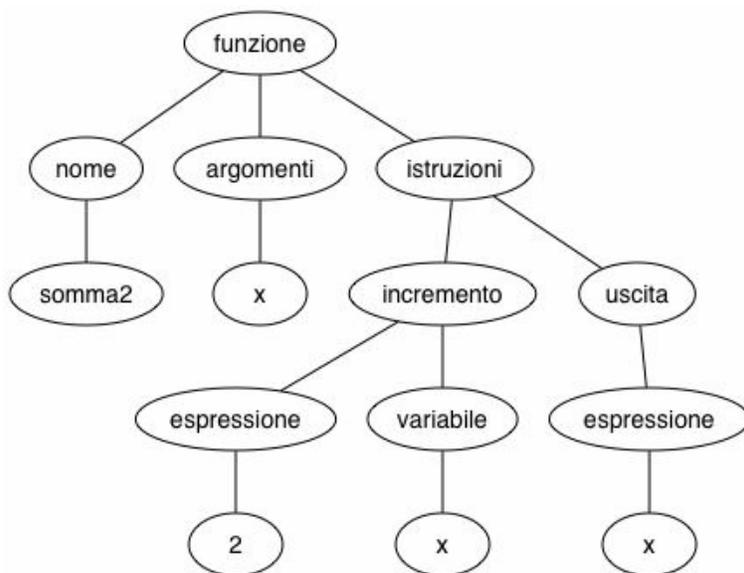
L'albero in figura ci permette anche di introdurre un po' della terminologia standard relativa agli alberi. Ogni elemento dell'albero è chiamato *nodo*, nella figura "organismo", "vegetale", "vertebrato", ecc. Le linee che collegano i nodi sono chiamate *archi*. Ogni arco va da un nodo *genitore* a un nodo *figlio*. Generalmente, il nodo genitore è disegnato più in alto o più a sinistra del nodo figlio. La relazione genitore-figlio è quella che determina la struttura gerarchica dell'albero. In alberi di classificazione, il nodo genitore rappresenta un concetto più generale mentre il nodo figlio una specializzazione di questo, ad es. "animale-mammifero". Il nodo in cima è chiamato *radice*, nel nostro esempio "organismo", mentre i nodi in fondo, cioè quelli senza figli, sono detti *foglie*, nella figura "cetaceo", "umano", ecc. I *discendenti* di un nodo sono i suoi nodi figli, i figli dei figli e così via fino alle foglie. Il *sottoalbero* di un nodo è l'albero che ha come radice il nodo e comprende tutti i suoi discendenti. Possiamo mostrare l'albero del file system dell'esempio precedente.



Gli alberi sono anche usati per rappresentare analisi grammaticali o sintattiche sia in linguaggi naturali sia in linguaggi artificiali come linguaggi di markup o linguaggi di programmazione. In quest'ultimo caso si parla di *alberi di parsing*. La loro costruzione è generalmente il primo passo intrapreso da un qualsiasi compilatore o interprete. Nel capitolo successivo ne vedremo la costruzione ed uso relativamente al linguaggio di markup HTML. Mostriamo qui un semplice esempio per la seguente funzione.

```

def somma2(x):
    x += 2
    return x
  
```



## 14.4 ALBERI DI OGGETTI

L'operazione fondamentale su un albero è la visita di tutti i suoi nodi che permette, ad esempio, di stamparli, contarli, di fare ricerche, ecc. Un modo generale per rappresentare gli alberi in linguaggi di programmazione orientati agli oggetti come Python è di definire un tipo per i nodi i quali contengono una lista di oggetti dello stesso tipo. Conoscendo l'oggetto che rappresenta la radice, si può arrivare a un qualsiasi altro nodo dell'albero seguendo opportunamente i link da genitore a figlio.

Vediamo come fare ciò per l'albero dei file e directory. Ogni nodo rappresenta o un file o una directory. Se rappresenta un file non ha figli se invece rappresenta una directory ha per figli i nodi che rappresentano tutti i file e directory contenuti direttamente nella directory.

```
class FSNode(object):
    def __init__(self, path):
        self.path = path
        self.content = [] # lista dei nodi figli
    def __str__(self):
        return 'FSNode("' + self.path + "')'
```

Per creare l'albero che rappresenta tutti i file e le directory contenute a qualsiasi livello in una data directory, definiamo una funzione che partendo dal percorso della directory radice crea un nodo di tipo `FSNode` per ogni file e directory esplorato ricorsivamente e quando trova una directory aggiunge al suo attributo `content` i nodi figli.

```
def gen_fstree(path):
    '''Genera l'albero partendo dal percorso path e
    ritorna il nodo radice'''
    node = FSNode(path)
    if os.path.isdir(path):
        for name in os.listdir(path):
            if name.startswith('.'): continue
```

```

        fullpath = os.path.join(path, name)
        node.content += [gen_fstree(fullpath)]
    return node

tree = gen_fstree('Informatica')
print(tree)
# Out: FSNode("Informatica")

```

Per stampare l'intero albero scriviamo una funzione che visita ricorsivamente i nodi e stampa il nome di ogni nodo che incontra. Per rispecchiare la struttura dell'albero il nome di ogni nodo è stampato con una indentazione proporzionale al suo livello nell'albero. Per fare ciò occorre che la funzione prenda anche un argomento che indica il livello del nodo corrente.

```

def print_fstree(node, level):
    '''Stampa l'albero con radice node'''
    # os.path.basename ritorna l'ultima componente
    print(' '*level + os.path.basename(node.path))
    # stampa ricorsivamente i sottoalberi dei nodi figli
    for child in node.content:
        print_fstree(child, level + 1)

print_fstree(tree, 0)
# Out: Informatica
# Out:   Hardware
# Out:     Cache.txt
# Out:     CPU.txt
# Out:     RAM.txt
# Out:   Software
# Out:     Linguaggi
# Out:       C.txt
# Out:       Java.txt
# Out:       JavaScript.txt
# Out:       Python.txt
# Out:     SistemiOperativi
# Out:       Linux.txt
# Out:       MacOS.txt
# Out:       Windows.txt

```

Definiamo ora una funzione che conta ricorsivamente i nodi dell'albero, che vengono calcolati, per ogni nodo, come la somma dei nodi contenuti nei figli più uno.

```
def count_fstree(node):
    '''Ritorna il numero di nodi dell'albero di
    radice root'''
    count = 1
    # per ogni nodo figlio,
    for child in node.content:
        # conta i nodi nel suo sottoalbero
        count += count_fstree(child)
    return count

print(count_fstree(tree))
# Out: 15
```

Ancora una volta è utile aggiungere qualche stampa per capire cosa succede.

```
def count_fstree_print(node, level):
    '''Ritorna il numero di nodi dell'albero di
    radice root'''
    print(' '*level + os.path.basename(node.path))
    count = 1
    for child in node.content:
        count += count_fstree_print(child, level+1)
    print(' '*level, '->', count)
    return count

print(count_fstree_print(tree, 0))
# Out: Informatica
# Out:   Hardware
# Out:     Cache.txt
# Out:       -> 1
# Out:     CPU.txt
# Out:       -> 1
# Out:     RAM.txt
# Out:       -> 1
# Out:     -> 4
# Out:   Software
```

```

# Out: Linguaggi
# Out: C.txt
# Out: -> 1
# Out: Java.txt
# Out: -> 1
# Out: JavaScript.txt
# Out: -> 1
# Out: Python.txt
# Out: -> 1
# Out: -> 5
# Out: SistemiOperativi
# Out: Linux.txt
# Out: -> 1
# Out: MacOS.txt
# Out: -> 1
# Out: Windows.txt
# Out: -> 1
# Out: -> 4
# Out: -> 10
# Out: -> 15
# Out: 15

```

Infine, definiamo una funzione che fa una ricerca nell'albero e ritorna una lista dei nodi che hanno un nome dato. Chiaramente anche questa funzione può essere implementata facendo una visita ricorsiva dell'albero.

```

def find_fstree(node, name):
    '''Ritorna una lista dei nodi dell'albero di
    radice root con nome name'''
    ret = []
    if os.path.basename(node.path) == name:
        ret += [node]
    # per ogni nodo figlio,
    for child in node.content:
        # cerca ricorsivamente nel suo sottoalbero
        ret += find_fstree(child, name)
    return ret

print(find_fstree(tree, 'Python.txt')[0])
# Out: FSNode("Informatica/Software/Linguaggi/Python.txt")

```

## 15 DOCUMENTI STRUTTURATI

I documenti strutturati si possono rappresentare come alberi i cui nodi definiscono le differenti sezioni del documento, che a loro volta contengono testo formattato. L'elaborazione di un documento strutturato richiede come primo passo il *parsing*, cioè l'analisi della struttura sintattica del documento. Il risultato del parsing è il cosiddetto *albero di parsing*, che visitato ricorsivamente permette di ottenere informazioni o di modificare il documento in vari modi. Questo capitolo mostrerà come fare il parsing e poi l'elaborazione di documenti HTML, il formato delle pagine Web.

## 15.1 HTML

Per formattare un testo in modo da poter indicare che, ad esempio, una parte di testo è un titolo o che va enfatizzata, si possono usare i *linguaggi di markup*. Quello usato per le pagine Web è il linguaggio *HTML*, o *HyperText Markup Language*. L'HTML usa dei marcatori chiamati *tag* che marciano parti di un documento, come ad esempio titoli, paragrafi, links a pagine o contenuti multimediali. Un documento HTML è letto da un web browser che interpreta i tag HTML e visualizza la pagina nel modo che ben conosciamo. Questo capitolo non è una guida HTML, ma introduce solamente le basi per mostrare come i documenti HTML possono essere elaborati in Python.

L'HTML usa marcatori, detti tag, che delimitano le parti del documento che hanno specifiche proprietà. Ad esempio, un paragrafo è indicato con

```
<p>Questo è un paragrafo.</p>
```

dove il tag `<p>` indica l'inizio del paragrafo e `</p>` la fine. Per indicare due paragrafi consecutivi, basta aggiungere altri paragrafi a seguire. Nell'esempio sotto ci avvaliamo anche del tag `<em>` per enfatizzare testo e del tag `<img>` per inserire un'immagine. Quest'ultima conterrà l'*attributo* `src` che punta al file immagine da inserire.

```
<p>Questo è un paragrafo.</p>
<p>Questo è un altro paragrafo.</p>
<p>Questo è un <em>paragrafo</em>.</p>
<p>Questa è una immagine: </p>
```

La struttura di un documento HTML è determinata dai tag. Ogni tag delimita una porzione del documento che può contenere testo, immagini, ecc. e altri tag. Quindi il documento può essere visto come un albero i cui nodi sono determinati dai tag. I nodi figli di un nodo corrispondono ai tag e alle parti di testo contenuti direttamente nella porzione delimitata dal tag del nodo. Ogni nodo tag comprende una porzione del documento che inizia con l'apertura

di un tag indicata dal nome del tag tra i caratteri `<` e `>`, ad es. `<em>`, e termina con la chiusura del tag indicata dal nome del tag preceduto da `'/'` e sempre tra i caratteri `<`, `>`, ad es. `</em>`. Inoltre i tag, e quindi anche i nodi, possono avere degli attributi che specificano varie proprietà. Gli attributi sono indicati nell'apertura del tag con il loro nome seguito da `=` e una stringa che ne specifica il valore. Gli attributi sono opzionali. Il formato generale di un nodo è quindi

```
<tag attributo1="valore1"...>contenuto</tag>
```

Non tutti i tag hanno una chiusura esplicita, come ad esempio il tag `img`. I tag che invece devono essere chiusi hanno generalmente un contenuto. Il contenuto di un tag e del relativo nodo è una lista di altri nodi che sono i suoi figli nell'albero. Non ci sono tag per delimitare esplicitamente il testo semplice, però conviene che le porzioni di testo siano comunque comprese in opportuni nodi dell'albero del documento. Ai nodi di testo noi assegneremo il tag di comodo `_text_`. La struttura generale di un documento HTML ha la seguente forma

```
<html>
<head>
  <!-- attributi del documento,
        come titolo, autore, etc. -->
</head>
<body>
  <!-- contenuto del documento che
        viene visualizzato -->
</body>
</html>
```

I web browser accettano anche frammenti di HTML, cioè solamente la parte contenuta nel nodo del tag `body`. Ma nel seguito assumeremo che i documenti HTML siano completi. L'HTML ha molti tag per marcare un documento. Indichiamo qui i più comuni:

- `<html>contenuto</html>`: il documento HTML
- `<body>contenuto</body>`: il contenuto del documento

- `<h1>titolo</h1>`: un titolo di livello 1
- `<h2>titolo</h2>`: un titolo di livello 2
- `<p>contenuto</p>`: un paragrafo
- `<em>contenuto</em>`: enfatizza il contenuto
- ``: un'immagine scaricata dall'indirizzo `indirizzo`
- `<a href="indirizzo">contenuto</a>`: un link all'indirizzo `indirizzo`

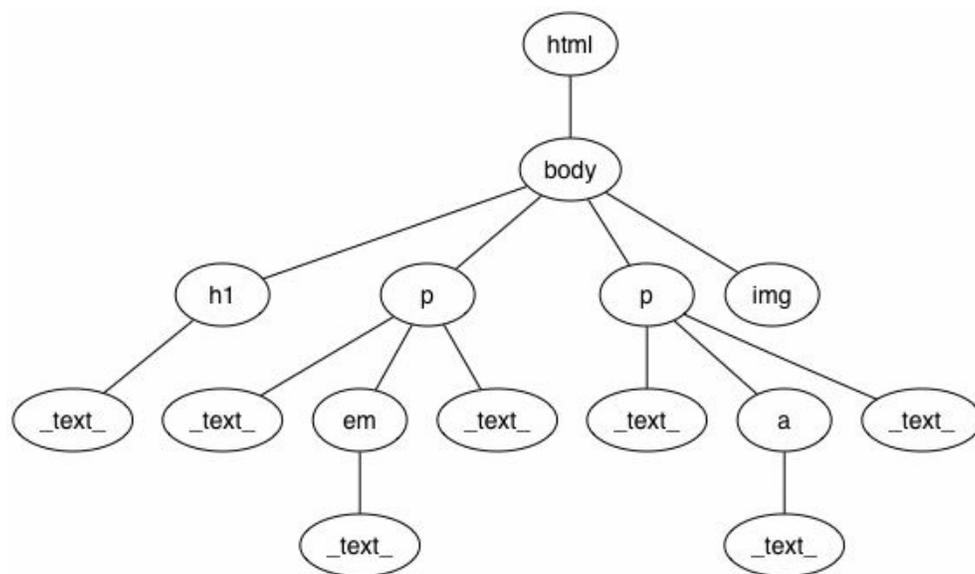
Un esempio molto semplice di documento HTML è il seguente:

```
<html>
<body>
<h1>Un Semplice Documento</h1>
<p>Un paragrafo con testo <em>enfattizzato</em>.</p>
<p>Un paragrafo con un link a <a
href="http://en.wikipedia.org/">Wikipedia</a> e un'immagine a
seguire.</p>

</body>
</html>
```



Visualizziamo ora la struttura ad albero del documento dove marcheremo i nodi di testo con `_text_` e non visualizzeremo il testo non visibile.



## 15.2 RAPPRESENTAZIONE DI DOCUMENTI HTML

Per rappresentare un documentato HTML useremo una classe che ne rappresenta i nodi con quattro variabili. `tag` è il nome del tag, ad esempio `'body'`. `attr` è il dizionario degli attributi, che rimarrà vuoto se l'elemento non ha attributi. Ogni chiave del dizionario è il nome di un attributo, di tipo stringa, a cui è associato il suo valore, di tipo stringa. Ad esempio `{ 'src': "img_logo.png" }`. `content` è il contenuto del nodo. Se il nodo è di testo, con tag `_text_`, il contenuto è la stringa di testo, altrimenti è una lista dei nodi figli, che può anche essere vuota. `closed` è `True` se il nodo ha la chiusura, altrimenti è `False`.

La classe conterrà metodi per stamparne il contenuto, `print_tree()` per visualizzare l'albero e `to_string()` che ritorna la stringa HTML corrispondente al nodo. Per quest'ultimo metodo dobbiamo considerare che HTML ha caratteri speciali per rappresentare i caratteri usati nei tags. Possiamo semplicemente usare la funzione `escape()` dal modulo `html` per supportare questo caso.

```
import html

class HTMLNode(object):
    def __init__(self, tag, attr, content, closed=True):
        self.tag = tag
        # dizionario degli attributi
        self.attr = attr
        # testo per nodi _text_ o lista dei figli
        self.content = content
        # True se il nodo ha la chiusura
        self.closed = closed

    # per distinguere i nodi testo
    def istext(self):
        return self.tag == '_text_'

    def open_tag(self):
```

```

'''Ritorna la stringa del tag di inizio.'''
if self.istext():
    return self.tag
s = '<'+self.tag
for k, v in self.attr.items():
    # usiamo escape per i valori
    s += ' {}="{}"'.format(
        k, html.escape(v,True))
s += '>'
return s

def close_tag(self):
'''Ritorna la stringa del tag di fine.'''
return '</'+self.tag+'>'

def print_tree(self, level=0):
'''Stampa l'albero mostrando la struttura
tramite indentazione'''
if self.istext():
    print(' '*level + '_text_ ' +
          repr(self.content))
else:
    print(' '*level + str(self))
    for child in self.content:
        child.print_tree(level+1)

def to_string(self):
'''Ritorna la stringa del documento HTML che
corrisponde all'albero di questo nodo.'''
if self.istext():
    # usiamo escape per i caratteri speciali
    return html.escape(self.content,False)
s = self.open_tag()
doc = self.open_tag()
if self.closed:
    for child in self.content:
        doc += child.to_string()
    doc += self.close_tag()
return doc

def __str__(self):
'''Ritorna una rappresentazione semplice
del nodo'''

```

```
if self.istext(): return self.tag
else: return '<{}>'.format(self.tag)
```

Possiamo quindi creare un piccolo documento HTML in modo programmatico.

```
doc = HTMLNode('html',{},{
    HTMLNode('body',{},{
        HTMLNode('p',{},{
            HTMLNode('_text_',{},{'Un paragrafo con '}),
            HTMLNode('em',{},{
                HTMLNode('_text_',{},{'enfasi'})
            ]),
            HTMLNode('_text_',{},{' e un\'immagine'}),
            HTMLNode('img',{ 'src':'img_logo.png'},
                [],closed=False)
        ]),
    ]),
])

# stampa la struttura nell'albero
doc.print_tree()
# Out: <html>
# Out:   <body>
# Out:     <p>
# Out:       _text_ 'Un paragrafo con '
# Out:       <em>
# Out:         _text_ 'enfasi'
# Out:         _text_ " e un'immagine"
# Out:         <img>

# stampa la stringa HTML corrispondente
print(doc.to_string())
# Out: <html><body><p>Un paragrafo con <em>enfasi</em> e
un'immagine</p></body></html>
```

## 15.3 PARSING DI DOCUMENTI HTML

Per elaborare un documento HTML la prima cosa da fare è costruire l'albero che ne rappresenta la struttura. Per fare ciò useremo il modulo `html.parser` della libreria standard ed in particolare la classe `HTMLParser`. Per creare il nostro parser dovremo modificare il comportamento della classe standard ridefinendone alcuni metodi. Python permette di fare ciò specificando una classe di base quando creiamo una nuova classe, con un meccanismo chiamato derivazione o *inheritance*. Non tratteremo questo argomento di programmazione in questo libro, se non usandolo in casi strettamente necessari, come la creazione di un parser HTML. Riportiamo a seguito il codice completo del parser per completezza, ma consigliamo al lettore di usarlo semplicemente come specificato.

```
import html.parser

class _MyHTMLParser(html.parser.HTMLParser):
    def __init__(self):
        '''Crea un parser per la class HTMLNode'''
        # inizializza la class base super()
        super().__init__()
        self.root = None
        self.stack = []
    def handle_starttag(self, tag, attrs):
        '''Metodo invocato per tag aperti'''
        closed = tag not in ['img', 'br']
        node = HTMLNode(tag, dict(attrs), [], closed)
        if not self.root:
            self.root = node
        if self.stack:
            self.stack[-1].content.append(node)
        if closed:
            self.stack.append(node)
    def handle_endtag(self, tag):
        '''Metodo invocato per tag chiusi'''
        if self.stack and self.stack[-1].tag == tag:
            self.stack[-1].opentag = False
```

```

        self.stack = self.stack[:-1]
def handle_data(self, data):
    '''Metodo invocato per il testo'''
    if not self.stack: return
    self.stack[-1].content.append(
        HTMLNode('_text_', {}, data))
def handle_comment(self, data):
    '''Metodo invocato per commenti HTML'''
    pass
def handle_entityref(self, name):
    '''Metodo invocato per caratteri speciali'''
    if name in name2codepoint:
        c = unichr(name2codepoint[name])
    else:
        c = '&'+name
    if not self.stack: return
    self.stack[-1].content.append(
        HTMLNode('_text_', {}, c))
def handle_charref(self, name):
    '''Metodo invocato per caratteri speciali'''
    if name.startswith('x'):
        c = unichr(int(name[1:], 16))
    else:
        c = unichr(int(name))
    if not self.stack: return
    self.stack[-1].content.append(
        HTMLNode('_text_', {}, c))
def handle_decl(self, data):
    '''Metodo invocato per le direttive HTML'''
    pass

```

Per usare la classe `_MyHTMLParser` basta crearne una istanza, chiamare il metodo `feed()` della classe di base e ritornare il contenuto della variabile `root`. Per semplicità definiamo le funzioni `parse()` e `fparse()` che ritornano l'albero dei nodi a partire rispettivamente da una stringa HTML o dal nome del file.

```

def parse(html):
    '''Esegue il parsing HTML del testo html e
    ritorna la radice dell'albero.'''
    parser = _MyHTMLParser()

```

```

parser.feed(html)
return parser.root

def fparse(fhtml):
    '''Esegue il parsing HTML del file fhtml e
    ritorna la radice dell'albero .'''
    with open(fhtml) as f:
        root = parse(f.read())
    return root

# Proviamo a fare il parsing del semplice file
# che abbiamo visto sopra.
doc = fparse('page_simple.html')

doc.print_tree()
# Out: <html>
# Out:  _text_ '\n'
# Out:  <body>
# Out:    _text_ '\n'
# Out:    <h1>
# Out:      _text_ 'Un Semplice Documento'
# Out:      _text_ '\n'
# Out:      <p>
# Out:        _text_ 'Un paragrafo con testo '
# Out:        <em>
# Out:          _text_ 'enfaticizzato'
# Out:          _text_ '.'
# Out:        _text_ '\n'
# Out:      <p>
# Out:        _text_ 'Un paragrafo con un link a '
# Out:        <a>
# Out:          _text_ 'Wikipedia'
# Out:          _text_ " e un'immagine a seguire."
# Out:        _text_ '\n'
# Out:      <img>
# Out:        _text_ '\n'
# Out:      _text_ '\n'

print(doc.to_string())
# Out: <html>
# Out: <body>
# Out: <h1>Un Semplice Documento</h1>
# Out: <p>Un paragrafo con testo <em>enfaticizzato</em>.</p>

```

```
# Out: <p>Un paragrafo con un link a <a  
href="http://en.wikipedia.org/">Wikipedia</a> e un'immagine a  
seguire.</p>  
# Out:   
# Out: </body>  
# Out: </html>
```

## 15.4 OPERAZIONI SUI DOCUMENTI

Ora che abbiamo l'albero di parsing possiamo facilmente fare delle operazioni su di esso. Possiamo implementare queste operazioni sia come metodi che come funzioni, che come si è potuto notare finora, sono molto simili nel nostro uso. Nel seguito scriveremo funzioni per semplicità.

Prima di tutto consideriamo operazioni che calcolano delle semplici statistiche, come contare il numero totale dei nodi e l'altezza dell'albero, quest'ultima definita come il numero di relazioni genitore-figlio massimale.

```
def count(node):  
    '''Ritorna il numero di nodi dell'albero di  
    questo nodo'''  
    cnt = 1  
    if not node.istext():  
        for child in node.content:  
            cnt += count(child)  
    return cnt
```

```
print('Numero di nodi:', count(doc))  
# Out: Numero di nodi: 22
```

```
def height(node):  
    '''Ritorna l'altezza dell'albero con radice  
    questo nodo, cioè il massimo numero di nodi  
    in un cammino radice-foglia'''  
    h = 1  
    if not node.istext():  
        for child in node.content:  
            h = max(h, height(child) + 1)  
    return h
```

```
print('Altezza:', height(doc))  
# Out: Altezza: 5
```

Poi possiamo aggiungere una funzione che ritorna una lista dei nodi che

hanno un dato tag.

```
def find_by_tag(node, tag):
    '''Ritorna una lista dei nodi che hanno il tag'''
    ret = []
    if node.tag == tag: ret += [node]
    if not node.istext():
        for child in node.content:
            ret += find_by_tag(child,tag)
    return ret

for node in find_by_tag(doc, 'a'):
    print(node.to_string())
# Out: <a href="http://en.wikipedia.org/">Wikipedia</a>
for node in find_by_tag(doc, 'p'):
    print(node.to_string())
# Out: <p>Un paragrafo con testo <em>enfaticizzato</em>.</p>
# Out: <p>Un paragrafo con un link a <a
href="http://en.wikipedia.org/">Wikipedia</a> e un'immagine a
seguire.</p>
```

Infine, implementiamo una funzione che modifica l'albero, ad esempio eliminando i nodi che hanno un tag dato. I nodi figli di un nodo eliminato diventano figli del nodo genitore del nodo eliminato.

```
def remove_by_tag(node, tag):
    '''Rimuove dall'albero tutti i nodi con il tag,
    esclusa la radice, cioè il nodo su cui è invocato
    il metodo.'''
    if node.istext(): return
    for child in node.content:
        remove_by_tag(child,tag)
    newcont = []
    for child in node.content:
        if child.tag == tag:
            if not child.istext():
                newcont += child.content
        else:
            newcont += [child]
    node.content = newcont
```

```

remove_by_tag(doc, 'a')
print(doc.to_string())
# Out: <html>
# Out: <body>
# Out: <h1>Un Semplice Documento</h1>
# Out: <p>Un paragrafo con testo <em>enfaticizzato</em>.</p>
# Out: <p>Un paragrafo con un link a Wikipedia e un'immagine a
seguire.</p>
# Out: 
# Out: </body>
# Out: </html>

remove_by_tag(doc, '_text_')
print(doc.to_string())
# Out: <html><body><h1></h1><p><em></em></p><p></p></body></html>

```

Finora abbiamo usato un documento d'esempio molto piccolo ma le pagine web sono di solito documenti HTML molto più complessi. Proviamo allora la nostra classe su documenti presi dal web, ad esempio calcolando qualche statistica sulla pagina iniziale del sito di Python o dalla pagina di Wikipedia del linguaggio Python.

```

from urllib.request import urlopen

def print_stats(url):
    '''Stampa alcune statistiche della pagina web
    all'url specificato.'''
    with urlopen(url) as f:
        page = f.read().decode('utf8')
        doc = parse(page)
        print('Numero di nodi:', count(doc))
        print('Altezza:', height(doc))
        print('Numero di links:', len(find_by_tag(doc, 'a')))
        print('Numero di immagini:', len(find_by_tag(doc, 'img')))

print_stats('http://python.org')
# Out: Numero di nodi: 1506
# Out: Altezza: 56
# Out: Numero di links: 196
# Out: Numero di immagini: 1

```

```
print_stats('https://en.wikipedia.org/wiki/Python_(programming_language)')
```

```
# Out: Numero di nodi: 9146
```

```
# Out: Altezza: 16
```

```
# Out: Numero di links: 1372
```

```
# Out: Numero di immagini: 22
```

## 16 INTERFACCE UTENTE

In questo capitolo introdurremo i concetti fondamentali per la creazione di interfacce utenti, creando un paio di semplici applicazioni interattive. Nei capitoli successivi useremo le interfacce utente come base per altri esempi di interazione.

## 16.1 PROGRAMMI INTERATTIVI

I programmi visti finora sono stati di tipo *non-interattivo* perché l'utente non poteva alterare il comportamento del programma durante la sua esecuzione. Invece un programma *interattivo* può cambiare il proprio comportamento in risposta all'input dell'utente, durante l'esecuzione. Tra i programmi interattivi ci sono quelli che rispondono solamente agli input da tastiera, i programmi grafici con bottoni e menu, e quelli che rispondono a interazioni "naturali" tramite touch screens. Gli ultimi due tipi usano interfacce utente grafiche, in inglese *Graphical User Interface*, o più brevemente *GUI*. Nel seguito considereremo le interfacce per sistemi a finestra, tipiche dei programmi su notebook e desktop.

## 16.2 LIBRERIE PER INTERFACCE UTENTE

Non ci sono librerie standard per le interfacce, sia per ragioni storiche che di mercato. Storiche perché sistemi differenti come Windows e Mac si sono evoluti adottando diverse filosofie di interfaccia utente e durante la loro evoluzione non hanno voluto perdere la compatibilità con i programmi precedenti. Le interfacce sono la parte del sistema operativo più visibile agli utenti e quindi per ragioni di mercato sono spesso usate per evidenziare le differenze tra i sistemi.

Ciò ha portato allo sviluppo di una moltitudine di librerie software per creare interfacce. Anche se molti dei principi di base sono simili, le librerie sono incompatibili tra loro. In questo libro utilizziamo **Qt 5**, una libreria multiplatforma per la creazione di interfacce. La libreria Qt tenta di uniformare il comportamento dei vari sistemi per agevolare lo sviluppo di GUI portabili. È una libreria vastissima sia come funzionalità sia come numero di funzioni e classi fornite. Non tenteremo di trattarla esaustivamente, la useremo solamente per illustrare i concetti fondamentali della programmazione delle GUI.

Qt è scritta nel linguaggio C++ per ragioni storiche e di efficienza. Per usare Qt in Python utilizzeremo un “wrapper” chiamato **PyQt 5**, che permettono di chiamare le funzioni della libreria C++ direttamente dal linguaggio Python. PyQt è installato di default da Anaconda. Per utilizzare PyQt5 useremo i seguenti imports.

```
# Importa tutte le classi per costruire GUI con Qt  
from PyQt5.QtCore import *  
from PyQt5.QtGui import *  
from PyQt5.QtWidgets import *  
from PyQt5.QtWebEngineWidgets import *
```

In generale, consigliamo di eseguire i programmi con GUI da terminale perché i programmi interattivi interferiscono con la shell di Python e

IPython.

## 16.3 APPLICAZIONI QT

In Qt l'interazione utente è gestita da un oggetto `QApplication` che prende il controllo del programma una volta che si fa partire l'applicazione. Possiamo creare vari elementi dell'interfaccia utente come finestre e bottoni che nella terminologia di Qt si chiamano *widget*. Iniziamo con la struttura base di un programma Qt con una finestra vuota.

```
# Crea un'applicazione Qt
app = QApplication([])

# Crea una finestra (ma non e' visibile)
window = QWidget()

# Imposta dimensione e titolo della finestra
window.resize(500, 300)
window.setWindowTitle('Fondamenti di Programmazione')

# Mostra la finestra
window.show()

# Lancia l'interazione con l'utente
app.exec_()
# Out: 0
```

Per creare un'applicazione, importiamo tutte le definizioni della libreria Qt dal modulo `PyQt5.QtWidgets` e creiamo un oggetto applicazione `QApplication` che amministra l'interazione con l'utente. Creiamo poi una finestra come oggetto della classe `QWidget` impostandone la dimensione e il titolo tramite i suoi metodi `resize` e `setWindowTitle`. Rendiamo poi la finestra visibile chiamando il metodo `show`. Infine, facciamo partire l'applicazione chiamando il metodo `exec_` dell'applicazione `app`. Da questo momento, il controllo del programma è completamente gestito da Qt che tornerà al Python solo quando la finestra sarà chiusa. Eseguendo il programma otteniamo la seguente applicazione, non molto interessante ma

già funzionale, la finestra è ridimensionabile e può essere chiusa.



## 16.4 METODI STATICI

Dato che utilizzeremo questa struttura per tutti gli esempi di questo capitolo, definiamo le funzioni `init()` per creare una finestra e `run_window()` per lanciare l'applicazione. Dato che Qt permette solo di avere una `QApplication`, useremo il metodo `QApplication.instance()` per determinare se una `QApplication` è già presente. Questo metodo è chiamato con la sintassi `NomeClasse.metodo`, cioè il metodo è chiamato in relazione alla classe e non a un suo oggetto. Metodi di questo tipo sono detti *metodi della classe* o anche *metodi statici* e sono usati spesso in linguaggi come il C++, con cui Qt è implementato. In Python questi metodi sono meno naturali e li troveremo quindi spesso relegati all'uso in casi di compatibilità con librerie di altri linguaggi.

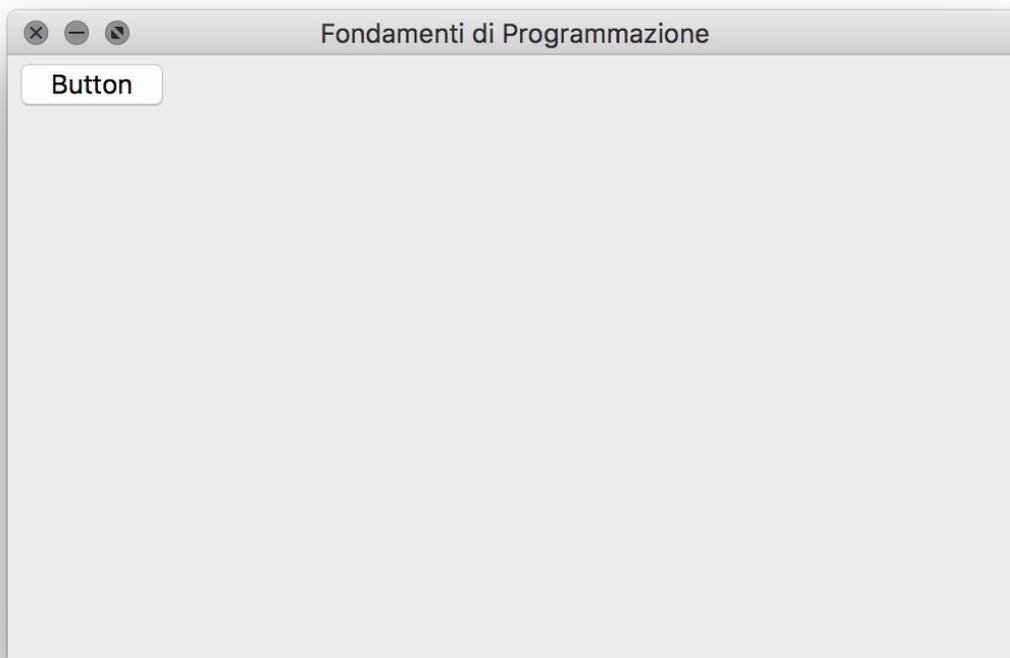
```
def init():
    '''Crea un'applicazione Qt e una finestra'''
    # verifica se l'applicazione è già esistente
    app = QApplication.instance()
    # o ne crea una nuova
    if not app: app = QApplication([])
    window = QWidget()
    window.resize(500, 300)
    window.setWindowTitle(
        'Fondamenti di Programmazione')
    return app, window

def run(app, window):
    '''Rende la finestra visibile e lancia
    l'applicazione'''
    window.show()
    app.exec_()
```

## 16.5 WIDGETS, EVENTI E CALLBACKS

Aggiungiamo ora dei componenti all'applicazione per renderla più utile. Possiamo aggiungere un bottone alla finestra semplicemente creandolo come oggetto della classe `QPushButton` dopo la creazione della finestra. Il primo argomento del costruttore è il testo che apparirà come titolo del bottone. Per attaccare il bottone alla finestra, passiamo l'oggetto `QWidget` della finestra, nel nostro caso `window`, come secondo argomento al costruttore di `QPushButton`.

```
# Crea un bottone sulla finestra  
app, window = init()  
button = QPushButton('Button', window)  
run(app, window)
```



Però cliccando sul bottone non succede nulla perché non abbiamo ancora specificato cosa fare in risposta ad un click. Per farlo utilizzeremo la *programmazione ad eventi*. L'idea principale è di specificare delle funzioni, dette *callback*, che sono chiamate da Qt quando l'utente genera *eventi*, ad esempio premendo il bottone.

Possiamo allora definire una semplice funzione `button_callback` che stampa un messaggio e collegarla tramite il metodo `connect()` all'evento di un mouse click sul bottone che è chiamato `clicked`. Definiamo ciò subito dopo la creazione del bottone e prima di lanciare l'applicazione. Al metodo `connect()` passiamo l'oggetto funzione, non il suo risultato della sua chiamata.

```
# Definisce la callback del bottone
def button_callback():
    print('Clicked')

app, window = init()

# Crea un bottone sulla finestra
button = QPushButton('Print', window)

# Imposta la callback in risposta all'evento
# clicked del bottone
button.clicked.connect(button_callback)

run(app, window)
# Out: Clicked
```

Ora, ogni volta che si clicca sul bottone, viene stampato il messaggio `Clicked`. Una callback può essere una funzione qualsiasi o un metodo, anche di un oggetto dell'interfaccia. Ad esempio, potremmo chiudere l'applicazione con un click sul bottone se impostiamo come callback il metodo `app.quit()`.



## 16.6 LAYOUTS

Per controllare come gli elementi di una interfaccia sono posizionati in una finestra, Qt usa il concetto di *layout*, cioè uno schema predefinito di disposizione. Ad esempio, se vogliamo aggiungere due bottoni disponendoli in verticale, creiamo un oggetto layout di disposizione verticale tramite la classe `QVBoxLayout` e aggiungiamo i due bottoni con il suo metodo `addWidget()`. Impostiamo poi il layout della finestra con il metodo `setLayout()`.

```
app, window = init()

# Crea un layout verticale
layout = QVBoxLayout()

# Crea due bottoni
button1 = QPushButton('Exit')
button2 = QPushButton('Print')

# Aggiunge i bottoni al layout
layout.addWidget(button1)
layout.addWidget(button2)

# Imposta il layout come layout della finestra
window.setLayout(layout)

# Definisce una semplice callback
def print_callback():
    print('ciao')

# Imposta le callback dei due bottoni
button1.clicked.connect(app.quit)
button2.clicked.connect(print_callback)

run(app, window)
```



## 16.7 ESEMPIO: TEXTEDITOR

Creeremo adesso un editor di testo minimale che avrà tre bottoni, uno per aprire un file di testo, uno per salvarlo e uno per uscire dall'applicazione, ed un'area per editare il testo, implementata grazie al widget `QTextEdit`. Per disporre questi elementi useremo un layout orizzontale `QHBoxLayout` per posizionare i tre bottoni come in una toolbar e poi un layout verticale `QVBoxLayout` per disporre la toolbar in alto e l'area per editare in basso. Il codice seguente crea gli elementi della nostra applicazione.

```
app, window = init()

# Crea i layout per la finestra e la toolbar
layout = QVBoxLayout()
tlayout = QHBoxLayout()

# Crea i tre bottoni
button_open = QPushButton('Open')
button_save = QPushButton('Save')
button_exit = QPushButton('Exit')

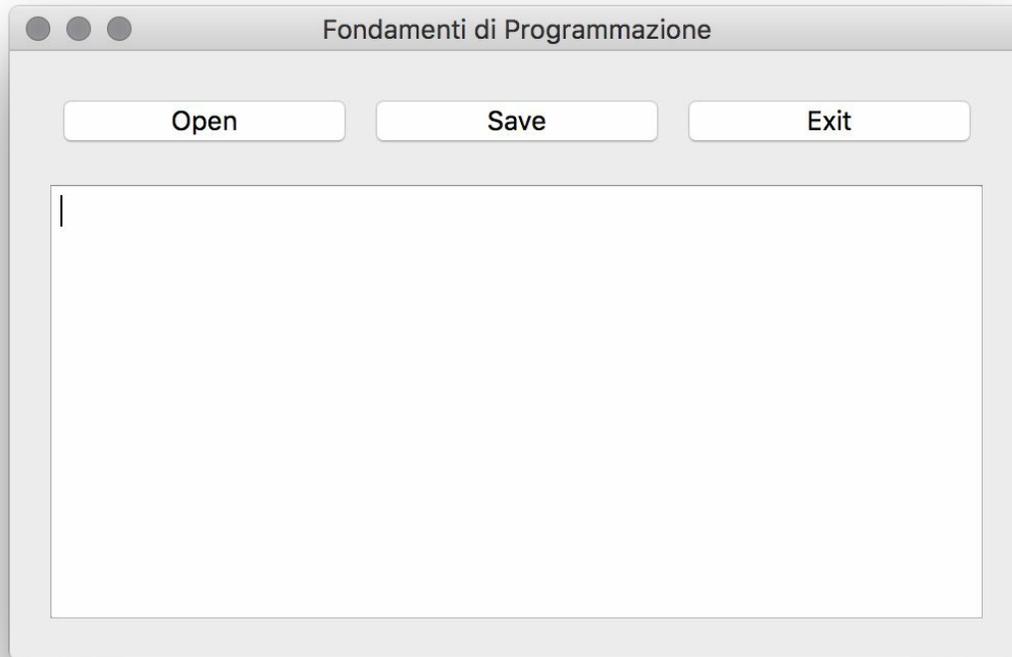
# Aggiunge i bottoni al layout della toolbar
tlayout.addWidget(button_open)
tlayout.addWidget(button_save)
tlayout.addWidget(button_exit)

# Aggiunge la toolbar al layout della finestra
layout.addLayout(tlayout)

# Crea un'area per editare testo
textedit = QTextEdit('')
# Aggiunge l'area testo al layout della finestra
layout.addWidget(textedit)

# Imposta il layout della finestra
window.setLayout(layout)
```

```
run(app, window)
```



Per caricare e salvare un testo definiamo due callbacks. Per selezionare un file, usiamo i *dialog box* offerti dalla classe `QFileDialog` di Qt. Useremo i metodi statici `getOpenFileName()` e `getSaveFileName()` che creano oggetti pre-configurati della classe `QFileDialog` e li rendono visibili in modo che l'utente possa scegliere un file. Alla fine dell'interazione, questi metodi ritornano il percorso del file scelto in una tupla il cui secondo argomento possiamo ignorare. Se l'utente sceglie di annullare l'operazione, i metodi ritornano la stringa vuota. Per interagire con l'area di testo `textedit` usiamo i metodi `setText()` per assegnare una stringa al testo visibile nella GUI, e `toPlainText()` per ottenere il contenuto dell'area di testo.

```
# Definisce la callback per aprire un file
def open_callback():
    filename, _ = QFileDialog.getOpenFileName(window)
    # Verifica se l'utente ha scelto "Cancel"
```

```

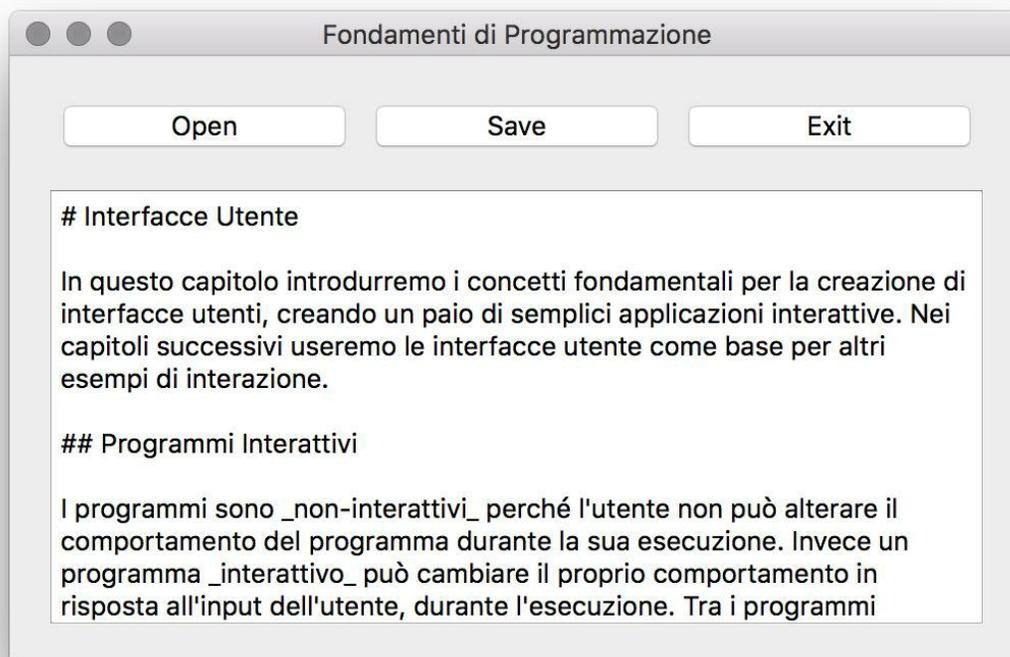
if not filename: return
with open(filename) as f:
    textedit.setText(f.read())

# Definisce la callback per salvare il file
def save_callback():
    filename, _ = QFileDialog.getSaveFileName(window)
    # Verifica se l'utente ha scelto "Cancel"
    if not filename: return
    with open(filename, 'w') as f:
        f.write(textedit.toPlainText())

# Imposta le callback dei tre bottoni
button_open.clicked.connect(open_callback)
button_save.clicked.connect(save_callback)
button_exit.clicked.connect(app.exit)

run(app, window)

```



## 16.8 ESEMPIO: WEBBROWSER

Come secondo esempio, creiamo un web browser minimale, usando l'engine Chromium incluso in Qt. Un oggetto della classe `QWebView`, dal modulo `PyQt5.QtWebEngineWidgets`, carica e visualizza una pagina web. Per visualizzare ed editare l'URL della pagina introdurremo una barra di navigazione tramite una linea di testo editabile implementata da `QLineEdit`. Aggiungeremo anche due bottoni per andare avanti ed indietro tra le pagine già scaricate. Il layout sarà simile all'esempio precedente.

```
app, window = init()

# Crea il layout per la finestra
layout = QVBoxLayout()

# Crea il layout per la toolbar
tlayout = QHBoxLayout()

# Crea la navigation bar e i bottoni di navigazione
text_bar = QLineEdit('')
button_back = QPushButton('<')
button_forward = QPushButton('>')

# Aggiunge i widgets alla toolbar
tlayout.addWidget(button_back)
tlayout.addWidget(button_forward)
tlayout.addWidget(text_bar)

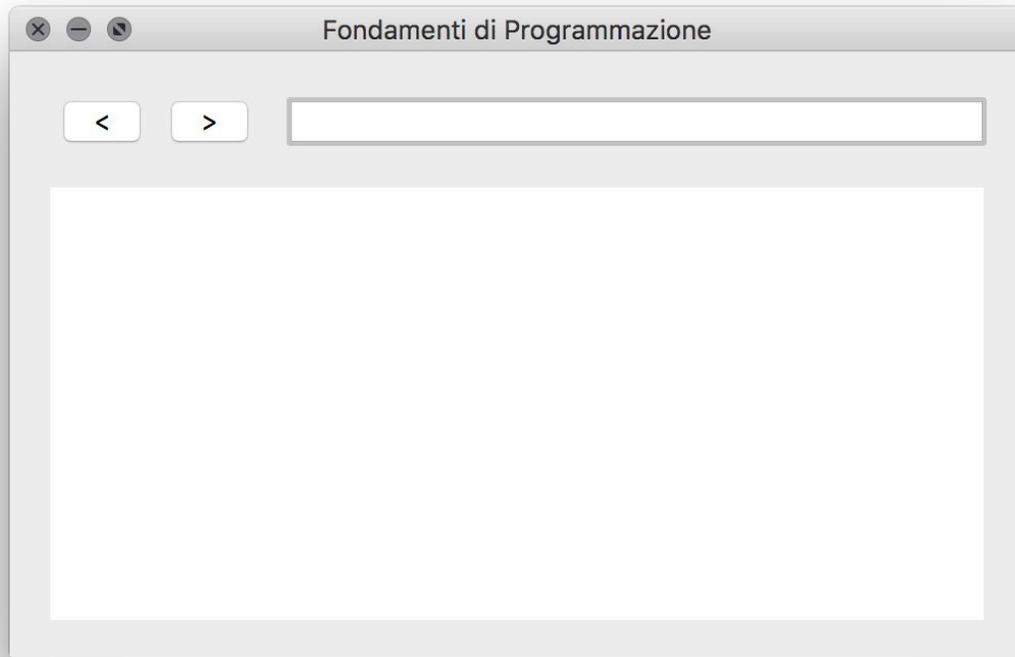
# Aggiunge la toolbar alla finestra
layout.addLayout(tlayout)

# Crea un widget per visualizzare pagine web
web_view = QWebView()
# Aggiunge al layout della finestra
layout.addWidget(web_view)

# Imposta il layout della finestra
```

```
window.setLayout(layout)
```

```
run(app,window)
```



Per interagire con la pagina impostiamo le seguenti azioni:

1. quando l'utente preme RETURN sulla `text_bar`, evento `returnPressed`, carichiamo nella `web_view` il nuovo url leggendolo dalla `text_bar`;
2. quando cambia l'url della pagina nella `web_view`, evento `urlChanged`, aggiorniamo il testo nella `text_bar`;
3. quando l'utente clicca sul bottone `back`, richiamiamo la pagina precedente della `web_view` tramite il suo metodo `back()`.
4. Quando l'utente clicca sul bottone `forward`, richiamiamo la pagina successiva della `web_view` tramite il suo metodo `forward()`.

Prima di tutto definiamo la callback per l'immissione di un nuovo URL da parte dell'utente. Il contenuto della `text_bar` si ottiene col metodo `text()` e si imposta l'URL da scaricare nella `web_view` col metodo `setUrl()` che però prende come argomento un oggetto `QUrl`.

Per aiutare l'utente l'URL può essere digitato senza lo schema. In tal caso, inseriamo lo schema `'http://'` automaticamente. Se il testo dell'utente non è interpretabile come indirizzo, ad esempio contiene spazi e non contiene punti, lo usiamo come interrogazione su Google. Aggiungeremo infine un test iniziale per evitare di ricaricare una pagina già caricata.

In risposta all'evento `urlChanged` di `web_view`, immettiamo il nuovo URL nella barra di navigazione impostandone il testo. Per i bottoni `back` and `forward`, chiameremo direttamente gli appropriati metodi di `web_view`.

```
# La callback per caricare una pagina
def load_page():
    if text_bar.text() == web_view.url().toString():
        return
    text = text_bar.text()
    if ' ' in text or '.' not in text:
        text = ('http://google.com/search?q=' +
               '+' .join(text.split()))
    elif '://' not in text:
        text = 'http://' + text
    web_view.setUrl(QUrl(text))

# La callback per il nuovo url nella navigation bar
def set_url():
    text_bar.setText(web_view.url().toString())

# Imposta la callback della navigation bar
text_bar.returnPressed.connect(load_page)
# Imposta la callback della web_view
web_view.urlChanged.connect(set_url)
# Imposta le callback dei due bottoni
button_back.clicked.connect(web_view.back)
button_forward.clicked.connect(web_view.forward)

run(app, window)
```



## 17 GRAFICA INTERATTIVA

In questo capitolo introdurremo alcuni semplici esempi di grafica interattiva che saranno il preludio alla creazione di applicazioni interattive più ricche.

## 17.1 SCHELETRO DELL'APPLICAZIONE

Adotteremo un modello molto semplice di grafica interattiva in cui forziamo l'applicazione ad aggiornare il proprio stato ogni sessantesimo di secondo, invece di attendere che si verifichi un evento. Il tempo di aggiornamento di un sessantesimo di secondo è sufficientemente breve per produrre animazioni convincenti. Per fare questo, dobbiamo creare un widget specifico modificando il comportamento di `QWidget` tramite derivazione. Dato che questo argomento di programmazione non è trattato in modo esplicito in questo libro, riportiamo il codice qui di seguito per completezza, consigliando al lettore di utilizzare il nuovo widget come se fosse un widget predefinito.

```
from PyQt5.QtCore import *
from PyQt5.QtGui import *
from PyQt5.QtWidgets import *

class PaintInfo:
    def __init__(self):
        self.mouse_pressed = False
        self.mouse_x = 0
        self.mouse_y = 0
        self.mouse_px = 0
        self.mouse_py = 0
        self.key = ''
        self.size = (0, 0)

class _GWidget(QWidget):
    def __init__(self):
        super().__init__()
        self.image = QImage(self.width(),
                             self.height(), QImage.Format_ARGB32)
        self.paint_handler = None
        self.info = PaintInfo()
        self.info.size = self.width(), self.height()
        self.setMouseTracking(True)
    def drawing(self):
```

```

painter = QPainter(self.image)
painter.setRenderHints(
    QPainter.Antialiasing,True)
painter.setRenderHints(
    QPainter.SmoothPixmapTransform,True)
painter.info = self.info
self.paint_handler(painter)
self.info.mouse_px = self.info.mouse_x
self.info.mouse_py = self.info.mouse_y
self.update()
def paintEvent(self, event):
    painter = QPainter(self)
    painter.drawImage(0, 0, self.image)
def resizeEvent(self,event):
    prev_img = self.image
    self.image = QImage(self.width(),
        self.height(), QImage.Format_ARGB32)
    self.image.fill(QColor(0,0,0).rgb())
    painter = QPainter(self.image)
    painter.drawImage(0, 0, prev_img)
    self.info.size = self.width(), self.height()
    self.update()
def mousePressEvent(self,event):
    self.info.mouse_pressed = True
def mouseReleaseEvent(self,event):
    self.info.mouse_pressed = False
def mouseMoveEvent(self,event):
    self.info.mouse_x = event.x()
    self.info.mouse_y = event.y()
def keyPressEvent(self,event):
    if not event.text():
        super().keyPressEvent(event)
    self.info.key = event.text()
def keyReleaseEvent(self,event):
    self.info.key = ''

def run_app(paint,w,h):
    app = QApplication.instance()
    if not app: app = QApplication([])
    widget = _GWidget()
    widget.resize(w,h)
    widget.setWindowTitle('Fondamenti di Programmazione')
    widget.paint_handler = paint

```

```
timer = QTimer()  
timer.setInterval(1000/60)  
timer.timeout.connect(widget.drawing)  
widget.show()  
timer.start()  
app.exec_()
```

Per usarlo basta chiamare la funzione `run_app(paint, w, h)` che crea l'applicazione Qt con una finestra di dimensioni `w` e `h` e che chiamerà la callback `paint()` ad ogni frame. Quest'ultima prende come argomento un oggetto `painter` di tipo `QPainter` che permette di disegnare varie forme semplici e che fornisce informazioni tramite l'oggetto interno `info`, di tipo `PaintInfo`, come le dimensioni della finestra. Il `painter` disegna su un'immagine che è poi copiata sullo schermo ad ogni frame.

## 17.2 DISEGNO DI FORME

Il `QPainter` usa un oggetto “penna” `QPen` per disegnare il contorno di una forma e un oggetto “pennello” `QBrush` per disegnarne l’interno. Questi possono essere impostati con i metodi `setPen()` e `setBrush()` di `QPainter`. I colori sono rappresentati da oggetti di tipo `QColor`. Quest’ultimo tipo è simile alla nostra classe `Color` ma, come si vedrà, è più versatile.

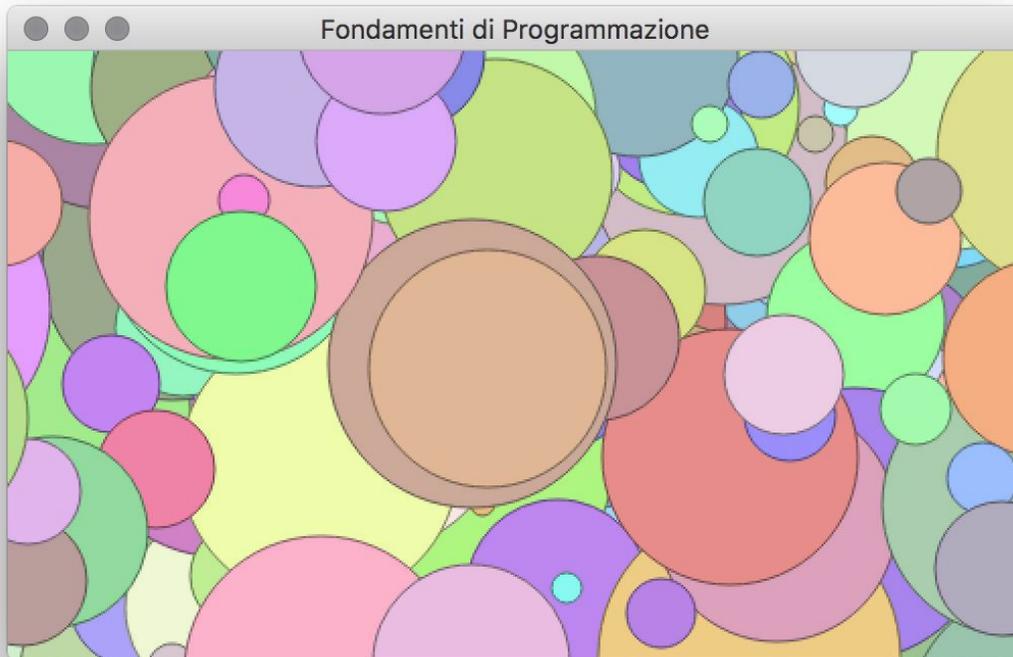
Iniziamo con esempi che disegnano ellissi `drawEllipse()`, rettangoli `drawRect()` e linee `drawLine()`. Ellissi e rettangoli sono specificati dalle coordinate dell’angolo in alto a sinistra e le dimensioni. Le linee sono specificate dalle coordinate di due punti. Ricordiamo che non cancellando l’immagine, le forme vengono disegnate una per frame sull’immagine corrente, quindi si sovrappongono.

```
from random import randint

def random_color(a=128,b=255):
    '''Colore casuale tra a e b'''
    return randint(a,b), randint(a,b), randint(a,b)

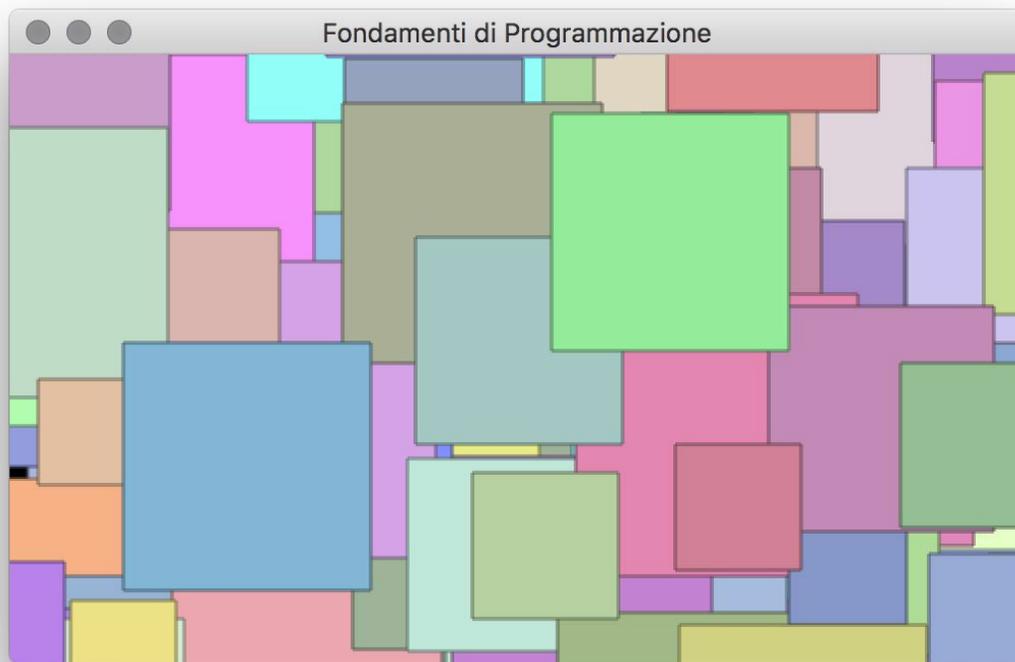
def paint(painter):
    '''Disegna un cerchio casuale'''
    # Dimensioni della finestra
    width, height = painter.info.size
    # Colore scelto in modo casuale
    r, g, b = random_color()
    # Colore del contorno e dell'interno
    painter.setPen(QColor(r//2,g//2,b//2))
    painter.setBrush(QColor(r,g,b))
    # Diametro e posizione casuali
    s = randint(10, 150)
    x = randint(-s, width+s)
    y = randint(-s, height+s)
    # Disegna il cerchio
    painter.drawEllipse(x, y, s, s)
```

```
run_app(Paint, 500, 300)
```



```
def paint(painter):  
    '''Disegna un rettangolo casuale'''  
    width, height = painter.info.size  
    r, g, b = random_color()  
    painter.setPen(QColor(r//2,g//2,b//2))  
    painter.setBrush(QColor(r,g,b))  
    # Lato e posizione del quadrato casuale  
    s = randint(50, 150)  
    x = randint(-s, width+s)  
    y = randint(-s, height+s)  
    # Disegna il quadrato  
    painter.drawRect(x, y, s, s)
```

```
run_app(Paint, 500, 300)
```



```
def paint(painter):  
    '''Disegna una linea casuale'''  
    width, height = painter.info.size  
    r, g, b = random_color()  
    # Spessore della linea casuale  
    lw = randint(1, 10)  
    painter.setPen(QPen(QColor(r,g,b), lw))  
    # Estremi della linea casuali  
    x1 = randint(-20, width+20)  
    y1 = randint(-20, height+20)  
    x2 = randint(-20, width+20)  
    y2 = randint(-20, height+20)  
    # Disegna la linea  
    painter.drawLine(x1, y1, x2, y2)  
  
run_app(paint, 500, 300)
```



Disegnare caratteri o testo è anch'esso molto semplice con il metodo `drawText()` di `QPainter`. Per cambiare la font si crea un oggetto `QFont` inizializzandolo con il nome e la dimensione del font che si vuole, ad esempio Helvetica a 12 punti, e poi lo si imposta con il metodo `setFont()`. Il `QPainter` userà il colore della penna per disegnare il testo.

```
def paint(painter):  
    '''Disegna un carattere casuale'''  
    width, height = painter.info.size  
    r, g, b = random_color()  
    # Dimensione del font casuale  
    fw = randint(12, 196)  
    # Imposta la fonte  
    painter.setFont(QFont('Helvetica', fw))  
    painter.setPen(QColor(r,g,b))  
    # Posizione random  
    x = randint(-20, width+20)  
    y = randint(-20, width+20)
```

```
# Carattere random
c = chr(randint(ord('A'),ord('z')))
# Disegna il carattere
painter.drawText(x, y, c)

run_app(paint,500,300)
```



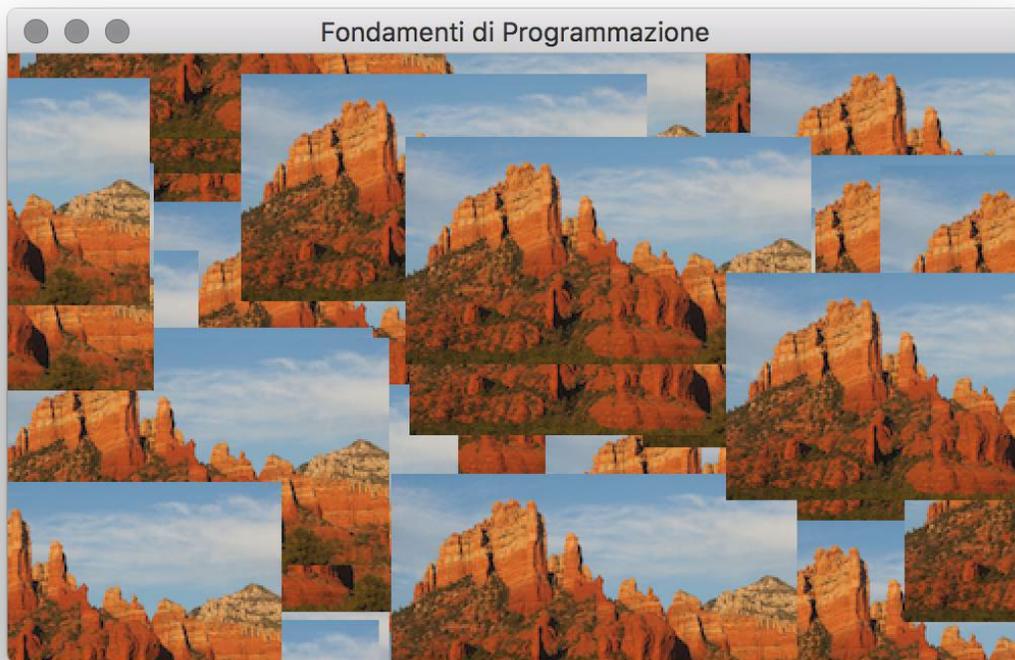
È altrettanto facile disegnare un'immagine. Basta caricare l'immagine in un oggetto `QImage` e disegnarla con il metodo `drawImage()`. I metodi `width()` e `height()` di `QImage` ritornano le dimensioni dell'immagine.

```
# Carica l'immagine da file
img = QImage('photo.png')

def paint(painter):
    '''Disegna l'immagine img in posizione casuale'''
    width, height = painter.info.size
    # Posizione casuale
```

```
x = randint(-img.width(), width+img.width())
y = randint(-img.height(), height+img.height())
# Disegna l'immagine img con l'angolo in
# alto a sinistra in (x,y)
painter.drawImage(x, y, img)
```

```
run_app(paint, 500, 300)
```



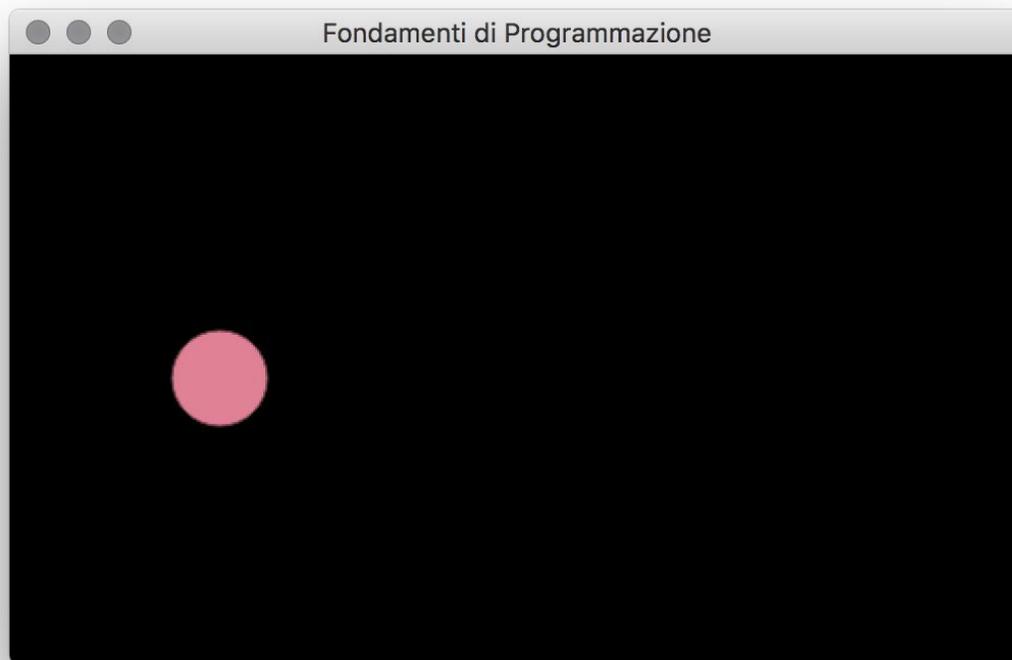
## 17.3 PULIZIA DELL'IMMAGINE

Finora abbiamo lasciato che i disegni dei frame si sovrapponevano l'uno sull'altro. Per evitare ciò possiamo ripulire l'immagine della finestra disegnando un rettangolo nero che copre l'intera immagine prima del disegno ad ogni frame. Se invece di disegnare un rettangolo nero solido, ne disegniamo uno semi-trasparente i disegni dei frame precedenti scompariranno progressivamente, lasciando tracce che diventeranno sempre più tenui. Il costruttore di `QColor` ammette anche un quarto canale, chiamato *alfa*, che indica il livello di trasparenza del colore. I suoi valori possono variare come per gli altri canali da 0 a 255, dove 0 significa completa trasparenza e 255 completa opacità.

```
def clear(painter,a=255):
    '''Pulisce l'immagine della finestra con
    trasparenza a'''
    width, height = painter.info.size
    painter.setPen(QColor(0,0,0,a))
    painter.setBrush(QColor(0,0,0,a))
    painter.drawRect(0, 0, width, height)

def paint(painter):
    '''Pulisce la finestra poi disegna'''
    clear(painter)
    width, height = painter.info.size
    r, g, b = random_color()
    painter.setPen(QColor(r//2,g//2,b//2))
    painter.setBrush(QColor(r,g,b))
    s = randint(10, 150)
    x = randint(-s, width+s)
    y = randint(-s, height+s)
    painter.drawEllipse(x, y, s, s)

run_app(paint,500,300)
```



```
def paint(painter):  
    '''Decolora la finestra prima di disegnare'''  
    # Pulisce la finestra parzialmente  
    clear(painter,4)  
    width, height = painter.info.size  
    r, g, b = random_color()  
    painter.setPen(QColor(r//2,g//2,b//2))  
    painter.setBrush(QColor(r,g,b))  
    s = randint(10, 150)  
    x = randint(-s, width+s)  
    y = randint(-s, height+s)  
    painter.drawEllipse(x, y, s, s)  
  
run_app(paint,500,300)
```



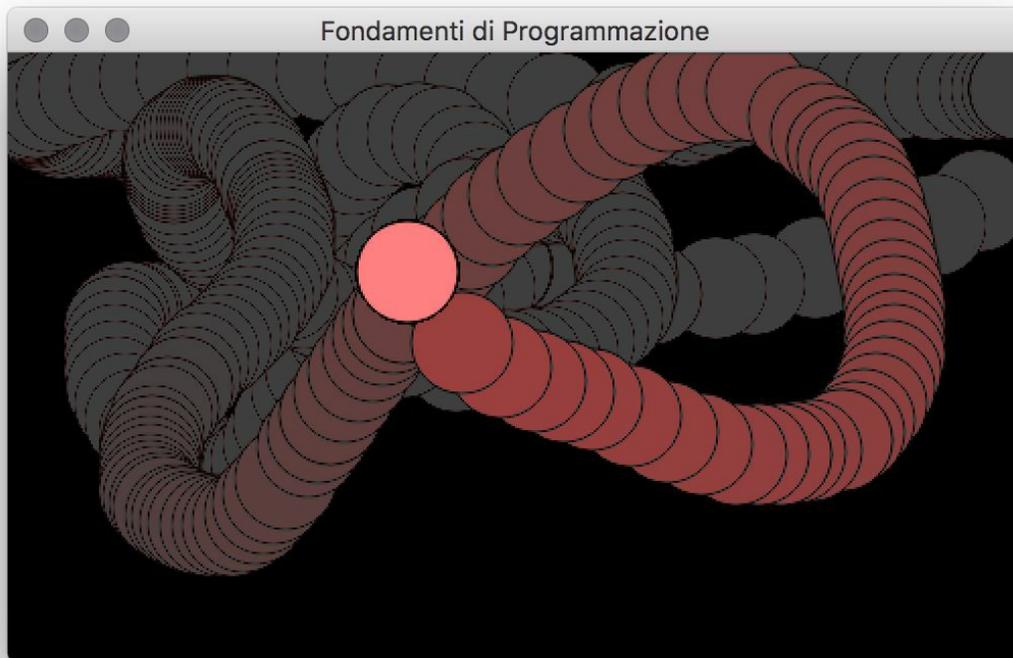
## 17.4 INTERAZIONE

Per interagire con il mouse e la tastiera, abbiamo aggiunto al `QPainter` un attributo `info` con valore un oggetto di tipo `PaintInfo` contenente i valori correnti di vari parametri tra cui quelli del mouse e della tastiera. In particolare, l'oggetto dice se un pulsante del mouse è premuto `mouse_pressed`, dà la posizione corrente del mouse `mouse_x` e `mouse_y`, la posizione precedente `mouse_px` e `mouse_py`, l'ultimo tasto premuto `key` e la dimensione dell'immagine `size`.

Iniziamo con il mouse disegnando cerchi centrati sulla sua posizione usando la dissolvenza progressiva per renderlo più interessante. Il colore del cerchio sarà deciso dalla pressione o meno del pulsante del mouse.

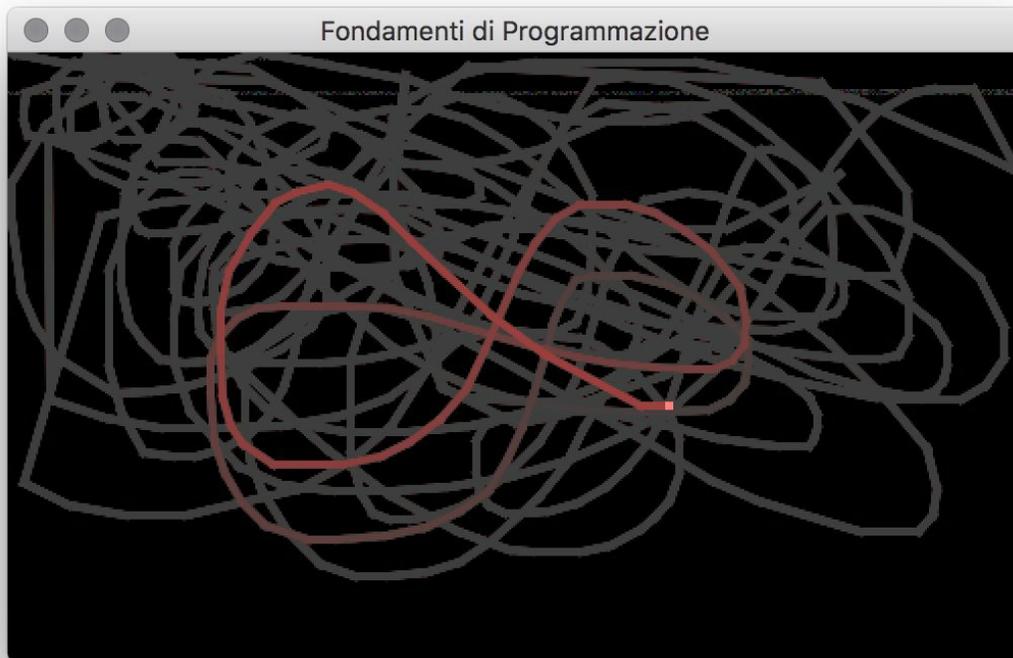
```
def paint(painter):
    '''Disegna un cerchio centrato nel mouse'''
    clear(painter, 2)
    painter.setPen(QColor(0,0,0))
    if painter.info.mouse_pressed:
        color = (128,255,128)
    else:
        color = (255,128,128)
    painter.setBrush(QColor(*color))
    painter.drawEllipse(painter.info.mouse_x - 25,
                       painter.info.mouse_y - 25, 50, 50)

run_app(paint,500,300)
```



Usando la posizione precedente del mouse, possiamo disegnare linee continue. Ad ogni frame, connettiamo semplicemente la posizione corrente con quella precedente.

```
def paint(painter):  
    '''Disegna una linea seguendo il mouse'''  
    clear(painter, 2)  
    if painter.info.mouse_pressed:  
        color = (128,255,128)  
    else:  
        color = (255,128,128)  
    painter.setPen(QPen(QColor(*color),4))  
    painter.drawLine(  
        painter.info.mouse_px,painter.info.mouse_py,  
        painter.info.mouse_x,painter.info.mouse_y)  
  
run_app(paint,500,300)
```



Infine, usiamo la tastiera per disegnare caratteri. Si noti che quando un tasto è rilasciato, il valore di `info.key` diventa la stringa vuota.

```
def paint(painter):  
    '''Disegna il carattere premuto sulla tastiera'''  
    clear(painter, 2)  
    if painter.info.mouse_pressed:  
        color = (128,255,128)  
    else:  
        color = (255,128,128)  
    painter.setPen(QColor(*color))  
    painter.setFont(QFont('Helvetica',300))  
    painter.drawText(painter.info.mouse_x,  
                    painter.info.mouse_y,painter.info.key)  
  
run_app(paint,500,300)
```



## 17.5 VARIABILI GLOBALI

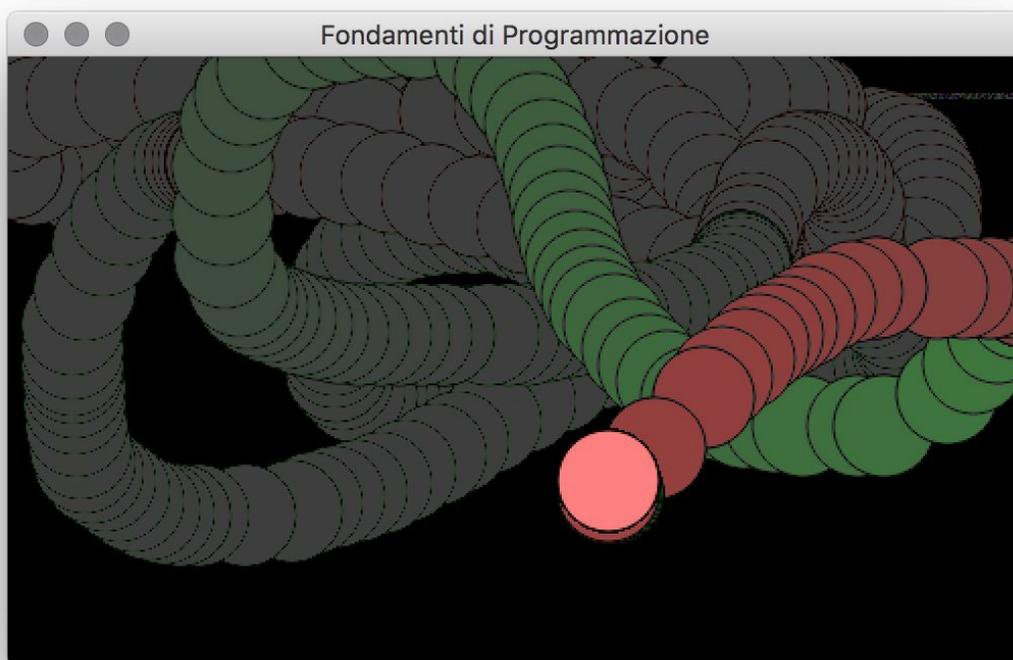
Possiamo scrivere un'applicazione che permette all'utente di scegliere le modalità di disegno interattivo precedenti associando a ognuna di esse un certo tasto. Usiamo una variabile globale `scribblemode` per registrare il carattere associato alla modalità di disegno corrente. Se usassimo una variabile locale il suo valore sarebbe perso alla chiamata successiva. Dichiarando la variabile come `global` nel corpo della funzione `paint` possiamo modificarne il valore nella funzione evitando così che venga definita automaticamente una variabile locale con lo stesso nome. Dopo aver impostato una modalità, tramite un opportuno tasto, cancelliamo `info.key` assegnandogli la stringa vuota. Questo è per evitare di ripetere l'impostazione della modalità quando un tasto è mantenuto premuto.

```
# Modalita' di disegno, default cerchi
scribblemode = 'c'

def paint(painter):
    '''Disegna forme controllate della tastiera'''
    # Per poter modificare il valore della variabile
    global scribblemode
    if painter.info.key in ['c','l','t','s']:
        scribblemode = painter.info.key
        painter.info.key = ''
    # Blocca il disegno e la dissolvenza
    if scribblemode == 's':
        return
    clear(painter, 2)
    if painter.info.mouse_pressed:
        color = (128,255,128)
    else:
        color = (255,128,128)
    # Cerchi
    if scribblemode == 'c':
        painter.setPen(QColor(0,0,0))
        painter.setBrush(QColor(*color))
        painter.drawEllipse(painter.info.mouse_x-25,
```

```
        painter.info.mouse_y-25,50,50)
# Linee
elif scribblemode == 'l':
    painter.setPen(QPen(QColor(*color),4))
    painter.drawLine(painter.info.mouse_px,
                    painter.info.mouse_py,
                    painter.info.mouse_x,
                    painter.info.mouse_y)
# Caratteri
elif scribblemode == 't':
    painter.setPen(QColor(*color))
    painter.setFont(QFont('Helvetica',300))
    painter.drawText(painter.info.mouse_x,
                    painter.info.mouse_y,painter.info.key)

run_app(paint,500,300)
```



## 17.6 TRASFORMAZIONI

La libreria Qt ha anche la possibilità di trasformare le forme disegnate specificando trasformazioni del sistema di coordinate del disegno. Quando specifichiamo le coordinate in Qt lo facciamo allo stesso modo delle immagini, cioè con l'origine nell'angolo in alto a sinistra. Possiamo cambiare questo riferimento spostandolo, ruotandolo e scalandolo. Questo ha l'effetto di spostare, ruotare o scalare il disegno. È utile, ad esempio, per disegnare un'immagine ruotata.

Iniziamo, disegnando un'immagine centrata nelle coordinate `x` e `y` del mouse e disegnando anche gli assi. Per centrare l'immagine su `(x,y)`, dobbiamo calcolare le coordinate dell'angolo in alto a sinistra sottraendo metà larghezza e metà altezza.

```
def paint(painter):
    '''Disegna un'immagine in (x, y), che per
    testare seguone il mouse'''
    clear(painter)
    width, height = painter.info.size
    x = painter.info.mouse_x
    y = painter.info.mouse_y
    painter.setPen(QPen(QColor(255,255,255),4))
    painter.drawLine(-width+x,y,width+x,y)
    painter.drawLine(x,-height+y,x,height+y)
    painter.drawImage(-img.width()/2+x,
                      -img.height()/2+y, img)

run_app(paint,500,300)
```

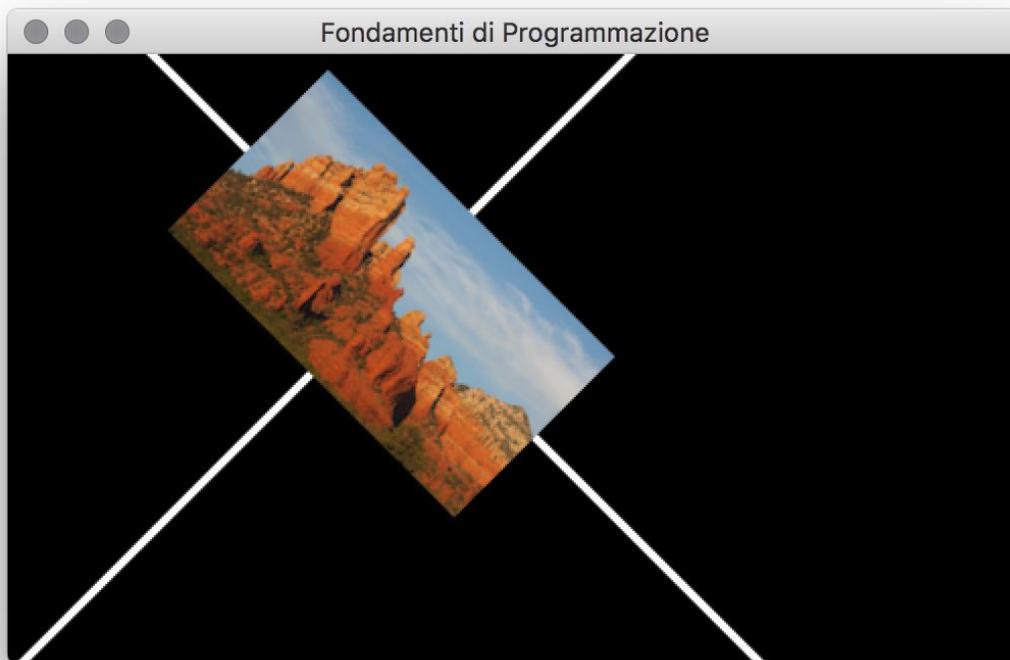


Per ottenere lo stesso effetto tramite una trasformazione del sistema di riferimento, possiamo fare una traslazione dell'intero disegno che porta l'origine in  $(x,y)$  e disegnare l'immagine centrata nell'origine. Possiamo adesso aggiungere una rotazione, ad esempio di 45 gradi.

```
def paint(painter):  
    '''Disegna un'immagine ruotata al centro'''  
    clear(painter)  
    width, height = painter.info.size  
    x = painter.info.mouse_x  
    y = painter.info.mouse_y  
    # Angolo della rotazione (in gradi)  
    a = 45  
    # Traslazione che porta l'origine in (x,y)  
    painter.translate(x,y)  
    # Rotazione intorno all'origine di angolo a  
    painter.rotate(a)  
    painter.setPen(QPen(QColor(255,255,255),4))  
    painter.drawLine(-width,0,width,0)
```

```
painter.drawLine(0,-height,0,height)
painter.drawImage(-img.width()/2,
                 -img.height()/2, img)

run_app(paint,500,300)
```



Possiamo ora prendere considerare posizioni e rotazioni casuali e aggiungere il fading.

```
def paint(painter):
    '''Disegna un'immagine con rotazione e
    posizione casuale'''
    clear(painter, 8)
    width, height = painter.info.size
    # Centro casuale
    x, y = randint(0,width), randint(0,height)
    # Angolo di rotazione casuale
    a = randint(0,360)
    painter.translate(x,y)
```

```
painter.rotate(a)  
painter.drawImage(-img.width()/2,  
-img.height()/2, img)
```

```
run_app(paint, 500, 300)
```



## 18 GIOCHI

In questo capitolo svilupperemo un semplice gioco per mostrare come si può usare la grafica interattiva in combinazione con l'interazione utente. Utilizzeremo la funzione `run_app()` e la classe `_GWidget` introdotte in precedenza, che per comodità considereremo salvate in un modulo `gwidget.py`.

## 18.1 STATO DEL GIOCO

Con gli strumenti introdotti in precedenza svilupperemo ora un piccolo gioco interattivo che è una versione semplice dello storico gioco **Tetris**. Non implementeremo il calcolo e la visualizzazione dei punteggi e assumiamo che il lettore conosce le regole del Tetris. Per utilizzare `gwidget` e Qt iniziamo col dichiarare alcuni imports.

```
from gwidget import run_app
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
```

Per prima cosa vediamo come rappresentare lo stato in cui si può trovare il gioco in un qualsiasi momento. Dobbiamo poter rappresentare i pezzi che sono già caduti e il pezzo che sta cadendo. Ogni pezzo è composto da quattro quadrati disposti in una certa forma. Per rappresentare la disposizione dei quadrati usiamo una matrice `board` i cui elementi sono i quadrati che possono essere vuoti, o occupati dal pezzo corrente o da quelli precedenti. Le dimensioni della matrice sono pari alle dimensioni dello spazio in cui i pezzi si muovono. Per rappresentare i colore dei quadrati usiamo stringhe che indicano i colori. Ad esempio, `'r'` rappresenta un quadrato di colore rosso. Per rappresentare una cella vuota, cioè una che non è occupata da un pezzo, usiamo la stringa vuota. In modo simile, definiamo ogni pezzo tramite una matrice di colori, o celle vuote, sufficientemente grande da contenere la forma del pezzo. Usiamo un dizionario `colors` per definire le associazioni tra le stringa e i colori Qt con cui disegnarle. Le forme possibili per i pezzi sono mantenute in una lista `pieces` contenente le matrici che definiscono tali forme.

```
# Dizionario dei colori
colors = { 'r': QColor(255,128,128),
           'g': QColor(128,255,128),
           'b': QColor(128,128,255),
```

```

        'c': QColor(128,255,255),
        'm': QColor(255,128,255),
        'y': QColor(255,255,128),
        'o': QColor(255,128,0) }

# Lista delle matrici delle forme dei pezzi
pieces = [
    [ ['c','c'], ['c',''], ['c',''] ],
    [ ['', 'r'], ['r','r'], ['r',''] ],
    [ ['o','o'], ['', 'o'], ['', 'o'] ],
    [ ['g',''], ['g','g'], ['', 'g'] ],
    [ ['b'], ['b'], ['b'], ['b'] ],
    [ ['m','m'], ['m','m'] ],
    [ ['y',''], ['y','y'], ['y',''] ]
]

```

Per completare la rappresentazione dello stato di gioco usiamo la variabile globale `piece` per contenere la matrice del pezzo corrente, cioè il pezzo che sta cadendo, e `piece_x`, `piece_y` per le sue coordinate definite rispetto alla matrice di gioco `board`. Inoltre dobbiamo mantenere un conteggio dei frame per sapere quando il pezzo corrente deve spostarsi verso il basso. Per questo usiamo la variabile globale `frame_count`. Per comodità manteniamo le dimensioni della matrice di gioco `board` nelle variabili globali `board_w`, `board_h` e quelle della matrice del pezzo corrente in `piece_w`, `piece_h`. Nella nostra implementazione usiamo una matrice di `22x10`, più larga dell'originale, per vedere meglio il gioco durante la stesura del codice.

```

# Dimensioni matrice di gioco
board_w, board_h = 22, 10
# Matrice di gioco, inizializzata
board = []
for _ in range(board_h):
    board.append( ['']*board_w )

# Forma del pezzo corrente
piece = pieces[0]
# Larghezza e altezza del pezzo corrente
piece_w = len(piece[0])
piece_h = len(piece)
# Posizione pezzo corrente

```

```
piece_x, piece_y = board_w//2, 0
# Conteggio frames per la caduta del pezzo
frame_count = 0
```

## 18.2 GRAFICA

La grafica del gioco è molto semplice. Per ogni elemento della matrice, disegniamo un quadrato, di lato `block_size`, col rispettivo colore. Definiamo una funzione `paint_blocks()` che poi useremo sia per disegnare la matrice di gioco che il pezzo corrente.

```
# Dimensione in pixel di un blocchetto
block_size = 16

def paint_blocks(painter, blocks, x, y, w, h):
    '''Disegna gli elementi della matrice blocks di
    dimensioni w, h, con l'angolo in alto a sinistra
    nel pixel di posizione x, y'''
    for j in range(h):
        for i in range(w):
            c = blocks[j][i]
            # Se non è una celletta vuota
            if not c: continue
            painter.setBrush(colors[c])
            painter.drawRect(
                (i+x)*block_size, (j+y)*block_size,
                block_size, block_size)
```

Nella funzione `paint()`, che sarà chiamata ad ogni frame, prima di aggiornare la grafica si dovrà gestire l'aggiornamento dello stato del gioco dovuto ad un eventuale tasto premuto e alla caduta del pezzo corrente. Per aggiornare lo stato del gioco definiremo fra poco una funzione `update()`.

```
def update(key):
    '''Aggiorna lo stato del gioco tenendo conto
    dell'eventuale tasto key'''
    # indica la funzione vuota
    pass

def paint(painter):
```

```

'''Aggiorna un frame del gioco'''
# Aggiorna lo stato del gioco
update(painter.info.key)
# Ripulisce la variabile della tastiera
painter.info.key = ''
# Ripulisci la finestra
painter.setBrush(QColor(128,128,128))
painter.drawRect(0, 0,
                 block_size*board_w, block_size*board_h)
# Disegna i pezzi
paint_blocks(painter, board, 0, 0,
            board_w, board_h)
# Disegna il pezzo corrente
if piece:
    paint_blocks(painter, piece,
                piece_x, piece_y, piece_w, piece_h)

# Crea la GUI (la finestra) e chiama la funzione
# paint() ad ogni frame
run_app(paint, board_w*block_size,
        board_h*block_size)

```



## 18.3 AGGIORNAMENTO DEL GIOCO

L'aggiornamento del gioco è definito nella funzione `update()`. Qui dobbiamo prima di tutto gestire l'eventuale tasto premuto dal giocatore e poi l'aggiornamento della caduta del pezzo corrente. Infatti il tasto premuto può modificare la posizione o rotazione del pezzo corrente e quindi anche la successiva caduta. Demandiamo la gestione del tasto ad una funzione `move()` che definiremo più avanti.

```
def move(key):
    '''Muovi (eventualmente) il pezzo corrente con
    tasto key, se possibile'''
    pass

def update(key):
    '''Aggiorna lo stato del gioco tenendo conto
    dell'eventuale tasto key'''
    # Se e' stato premuto un tasto,
    if key:
        # gestisci il tasto premuto
        move(key)
    # aggiornamento caduta del pezzo corrente ...
    pass
```

Durante il gioco, il pezzo corrente cade di una singola posizione, cioè di un quadrato, ogni volta che il numero di frames `frames_droppiece` è passato. Il conteggio dei frame è mantenuto in `frame_count` e viene azzerato ogni volta che il pezzo cade di una posizione. Notiamo però che non sappiamo se c'è spazio per far scendere il pezzo perché potrebbe aver raggiunto il fondo o potrebbe toccare altri pezzi già caduti. In questo caso, il pezzo viene congelato nella `board` e un altro pezzo è fatto partire.

Per implementare questa logica, proviamo a far scendere il pezzo con `piece_y += 1` e testiamo se collide con qualcosa tramite la funzione `hit()`. Se collide, torniamo alla posizione precedente e congeliamo il pezzo nella

`board` tramite la funzione `resolve_board()`, che eliminerà anche le eventuali righe piene, e iniziamo a far cadere un nuovo pezzo tramite la funzione `newpiece()`. Quest'ultima operazione potrebbe però fallire, se non c'è spazio per nessun nuovo pezzo. In Tetris questo è *game over*. Ancora una volta usiamo la funzione `hit()` per validare se il pezzo è ok e se non lo è, iniziamo di nuovo il gioco con la funzione `start()`. Modifichiamo ora l'implementazione di `update()` per includere le azioni descritte.

```
# Numero frames per caduta del pezzo
frames_droppiece = 30
# Frame corrente
frame_count = 0

def hit():
    '''Ritorna True se il pezzo corrente collide'''
    pass

def resolveboard():
    '''Aggiorna la matrice di gioco aggiungendo il
    pezzo corrente, che è arrivato, ed elimina le
    eventuali righe piene.'''
    pass

def newpiece():
    '''Crea un nuovo pezzo'''
    pass

def start():
    '''Inizializza la matrice di gioco vuota e crea
    un nuovo pezzo'''
    pass

def update(key):
    '''Aggiorna lo stato del gioco tenendo conto
    dell'eventuale tasto key'''
    if key:
        move(key)
    global piece_x, piece_y, frame_count
    # Incrementa il conteggio dei frames
    frame_count += 1
    # fai cadere il pezzo solo se sono passati
```

```
# frames_droppiece frames
if frame_count < frames_droppiece:
    return
# Muovi il pezzo corrente in basso
piece_y += 1
# Se adesso il pezzo collide,
if hit():
    # riportalo indietro e aggiorna il gioco
    piece_y -= 1
    resolveboard()
    # Crea un nuovo pezzo
    newpiece()
    # Se già collide, game over
    if hit():
        start()
frame_count = 0

piece_x, piece_y = board_w//2, 0
run_app(paint, board_w*block_size,
        board_h*block_size)
```



La funzione `hit()` controlla se un c'è almeno un quadrato del pezzo corrente che si sovrappone ad un quadrato nella matrice di gioco, o se il pezzo è uscito dall'area di gioco. La funzione `resolveboard()` copia il pezzo corrente nella matrice di gioco e poi elimina le eventuali righe piene facendo cadere le righe superiori.

```
def hit():
    '''Ritorna True se il pezzo corrente collide'''
    # Collisione bordi verticali
    if not (0 <= piece_x <= board_w-piece_w):
        return True
    # Collisione bordi orizzontali
    if not (0 <= piece_y <= board_h-piece_h):
        return True
    # Controlla se collide coi pezzi già caduti
    for j in range(piece_h):
        for i in range(piece_w):
            if (piece[j][i] and
                board[j+piece_y][i+piece_x]):
                return True
    return False

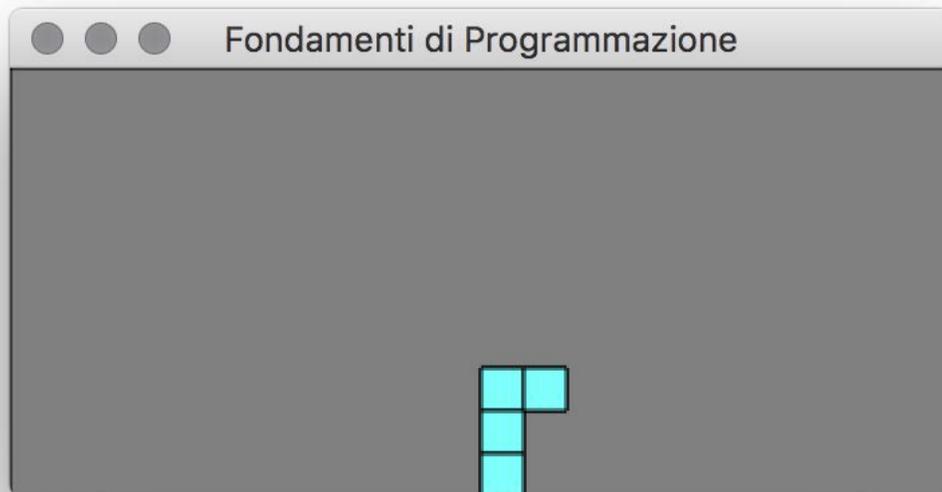
def resolveboard():
    '''Aggiorna la matrice di gioco aggiungendo il
    pezzo corrente, che è arrivato, ed elimina le
    eventuali righe piene.'''
    # Aggiungi il pezzo alla matrice di gioco
    for j in range(piece_h):
        for i in range(piece_w):
            if piece[j][i]:
                pj, pi = j+piece_y, i+piece_x
                board[pj][pi] = piece[j][i]
    # Cerca se ci sono righe piene da eliminare
    for j in range(board_h):
        # Se la riga j e' piena,
        if all(board[j]):
            # fai cadere le righe superiori
            for jj in range(j,0,-1):
                for ii in range(board_w):
                    board[jj][ii] = board[jj-1][ii]
            # e vuota la prima riga
```

```

        for ii in range(board_w):
            board[0][ii] = ' '

piece_x, piece_y = board_w//2, 0
run_app(paint, board_w*block_size,
        board_h*block_size)

```



Infine la funzione `start()` inizializza il gioco vuotando ogni cella della `board` e con `newpiece()` inizializza un nuovo pezzo scelto a caso, facendolo cadere dall'alto e centrato.

```

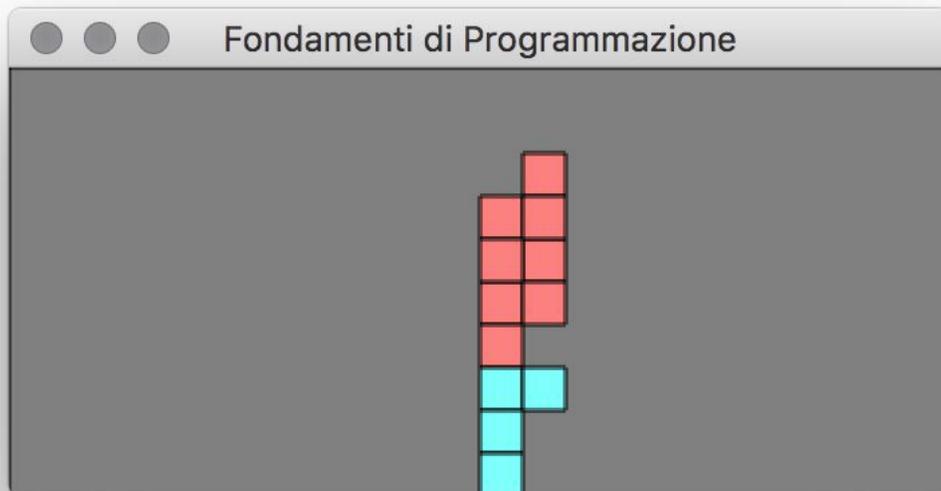
from random import choice

def newpiece():
    '''Crea un nuovo pezzo'''
    global piece, piece_x, piece_y, piece_w, piece_h
    # Scegli in modo casuale il nuovo pezzo
    piece = choice(pieces)
    # Imposta la posizione iniziale del pezzo
    piece_x, piece_y = board_w//2, 0
    piece_w, piece_h = len(piece[0]), len(piece)

```

```
def start():
    '''Inizializza la matrice di gioco vuota e crea
    un nuovo pezzo'''
    for j in range(board_h):
        for i in range(board_w):
            board[j][i] = ''
    newpiece()

start()
run_app(paint, board_w*block_size,
        board_h*block_size)
```



## 18.4 INTERAZIONE

L'interazione con il giocatore è implementata nella funzione `move()`, che per prima cosa tenta di muovere il pezzo corrente secondo il tasto premuto e poi testa se si è verificata una collisione con `hit()`. In questo caso, il movimento è invertito per tornare allo stato iniziale. Tra le mosse contemplate, ci sono lo spostamento orizzontale, tasti `a` e `d`, e verticale, `w` e `s`, implementate modificando la posizione del pezzo `piece_x`, `piece_y`. Lo spostamento in alto è utile durante lo sviluppo. La caduta rapida, (spazio, è implementata scendendo fino a quando non avviene una collisione. Includiamo anche il restart del gioco col tasto `g`. Infine, il pezzo può essere ruotato, tasti `q` e `e`. Per le rotazioni prevediamo due funzioni `rotater()` e `rotatel()` che saranno implementate a breve.

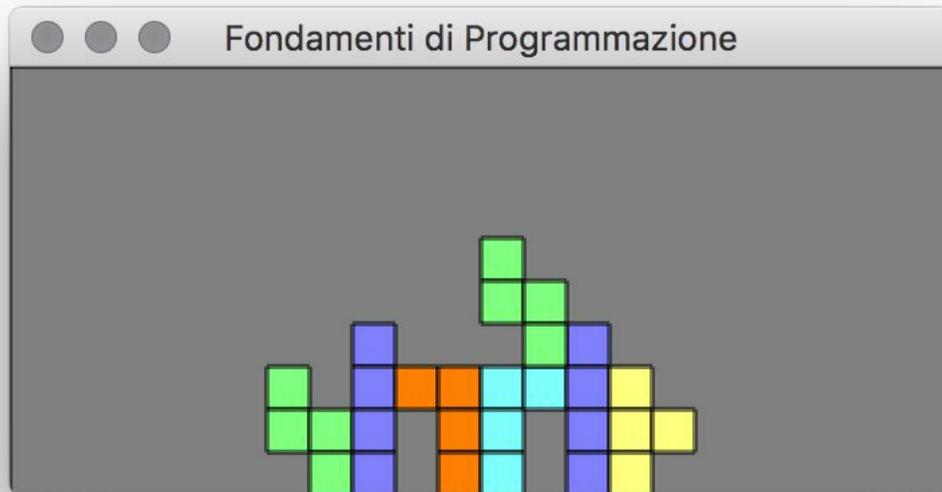
```
def rotater():
    '''Ruota a destra il pezzo corrente'''
    pass

def rotatel():
    '''Ruota a sinistra il pezzo corrente'''
    pass

def move(key):
    '''Muovi (eventualmente) il pezzo corrente con
    tasto key, se possibile'''
    global piece_x, piece_y
    # A sinistra
    if key == 'a':
        piece_x -= 1
        if hit(): piece_x += 1
    # A destra
    elif key == 'd':
        piece_x += 1
        if hit(): piece_x -= 1
    # In alto
    elif key == 'w':
        piece_y -= 1
```

```
        if hit(): piece_y += 1
# In basso
elif key == 's':
    piece_y += 1
    if hit(): piece_y -= 1
# Rotazione a sinistra
elif key == 'q':
    rotatel()
    if hit(): rotater()
# Rotazione a destra
elif key == 'e':
    rotater()
    if hit(): rotatel()
# Caduta immediata
elif key == ' ':
    while not hit(): piece_y += 1
    piece_y -= 1
# Inizia un nuovo gioco
elif key == 'g':
    start()
```

```
start()
run_app(paint, board_w*block_size,
        board_h*block_size)
```



Le rotazioni sono implementate in modo simile a quelle delle immagini, viste in precedenza. La modifica che facciamo è di invertire le coordinate `x` e `y` nelle due funzione per implementare rotazioni di 90 gradi o sinistra e destra.

```
def rotater():
    '''Ruota a destra il pezzo corrente'''
    global piece, piece_w, piece_h
    # Crea e inizializza vuota la matrice
    newp = []
    # per il pezzo ruotato
    for _ in range(piece_w):
        newp.append( ['']*piece_h )
    # Riempi la matrice con i valori ruotati
    for j in range(piece_h):
        for i in range(piece_w):
            newp[i][j] = piece[piece_h-1-j][i]
    piece_w, piece_h = piece_h, piece_w
    piece = newp

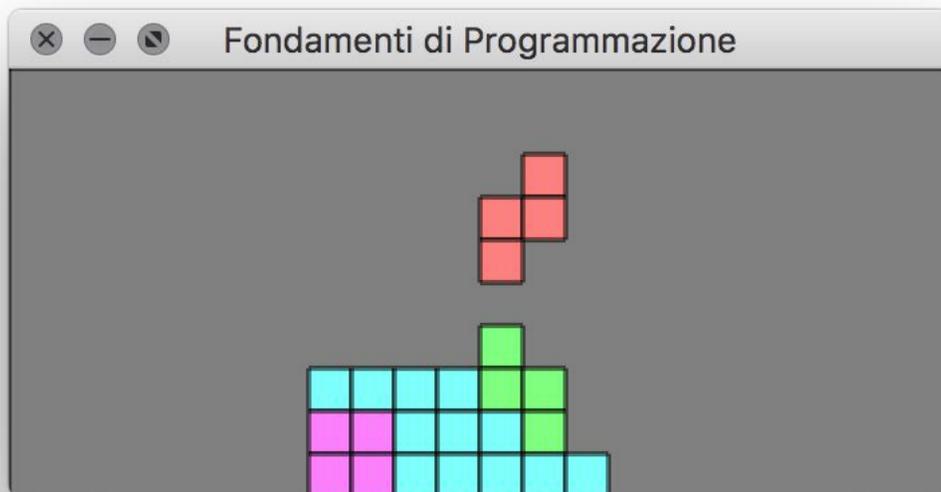
def rotatel():
    '''Ruota a sinistra il pezzo corrente'''
```

```

global piece, piece_w, piece_h
# Crea e inizializza vuota la matrice
newp = []
for _ in range(piece_w):
    newp.append( ['']*piece_h )
# Riempi la matrice con i valori ruotati
for j in range(piece_h):
    for i in range(piece_w):
        newp[i][j] = piece[j][piece_w-1-i]
piece_w, piece_h = piece_h, piece_w
piece = newp

start()
run_app(Paint, board_w*block_size,
        board_h*block_size)

```



## 19 SIMULAZIONE INTERATTIVA

In questo capitolo, vedremo le basi per creare simulazioni interattive prendendo come esempio il moto di un insieme di particelle soggette a vari tipi di forze in uno spazio bidimensionale. Questo tipo di simulazioni sono basilari per giochi ispirati alla fisica e per interfacce naturali come lo scrolling negli smartphone. Useremo ancora una volta il modulo `gwidget`.

## 19.1 SIMULAZIONE DI PARTICELLE

Svilupperemo un piccolo simulatore basato su principi fisici per muovere delle particelle in uno spazio bidimensionale. Nella simulazione la fisica sarà molto semplificata per poter implementare effetti interessanti con poco codice. Ogni particella ha una posizione  $p$ , con due componenti  $x$  e  $y$ , e una velocità  $v$ . Ad ogni frame dell'animazione la posizione è aggiornata in base alla velocità. A sua volta la velocità è aggiornata applicando forze che causano un'accelerazione  $a$  calcolata come forza diviso massa  $a=F/m$ . In ogni frame, i nuovi valori di posizione  $p'$  e velocità  $v'$  sono calcolati con il metodo di Eulero, che scriviamo in modo semplificato come  $v'=v+F/m$  e  $p'=p+v'$ , assumendo che l'intervallo di tempo tra un frame e l'altro è fisso e nominalmente 1.

## 19.2 VETTORI

Per modellare questi concetti, introduciamo la classe `vec2f` che rappresenta in modo conveniente punti e direzioni in due dimensioni, e le principali operazioni vettoriali su di essi. Questo permette di evitare di ripetere ogni operazione sulle due componenti.

```
from random import random, uniform, randint
from math import pi, sqrt, sin, cos

class vec2f(object):
    '''Vettore bidimensionale di float'''
    def __init__(self,x,y):
        self.x = float(x)
        self.y = float(y)
    def __add__(self,other):
        '''Somma di vettori'''
        return vec2f(self.x+other.x,self.y+other.y)
    def __sub__(self,other):
        '''Somma con vettore opposto'''
        return vec2f(self.x-other.x,self.y-other.y)
    def __neg__(self):
        '''Vettore opposto'''
        return vec2f(-self.x,-self.y)
    def __mul__(self,other):
        '''Prodotto per scalare'''
        return vec2f(self.x*other,self.y*other)
    def __truediv__(self,other):
        '''Divisione per scalare'''
        return vec2f(self.x/other,self.y/other)
    def length(self):
        '''Lunghezza del vettore'''
        return sqrt( self.x*self.x + self.y*self.y )
    def normalized(self):
        '''Vettore normalizzato'''
        l = self.length()
        if l < 0.000001:
            return vec2f(0,0)
```

```

        else:
            return vec2f(self.x/l,self.y/l)
def clamped(self,maxlen):
    '''Vettore con lunghezza massima maxlen'''
    l = self.length()
    if l > maxlen:
        return self * maxlen / l
    else:
        return vec2f(self.x,self.y)

# funzioni utili su vettori
def length(v): return v.length()
def normalize(v): return v.normalized()
def clamp(v,maxlength): return v.clamped(maxlength)

def dot(v0,v1):
    '''Prodotto scalare'''
    return v0.x*v1.x+v0.y*v1.y

def random_pos(x0, y0, x1, y1):
    '''Vettore random tra i valori dati'''
    return vec2f(uniform(x0,x1), uniform(y0,y1))

def random_dir():
    '''Vettore random di lunghezza 1'''
    a = uniform(0,2*pi)
    return vec2f(cos(a),sin(a))

def random_vec(maxlen):
    '''Vettore random di lunghezza al più maxlen'''
    return random_dir() * random() * maxlen

```

## 19.3 MODELLO PER LE PARTICELLE

Svilupperemo il programma della simulazione attraverso il graduale arricchimento di versioni intermedie. Partiremo da una sola particella immobile, ed arriveremo alla simulazione di un insieme di particelle il cui moto è soggetto a varie forze e all'interazione tra le particelle. Lo stato di una particella è mantenuto da oggetti della classe `Particle`, che contiene sia variabili per lo stato fisico, come la posizione, la velocità, l'accelerazione, il raggio, e la massa, che variabili di simulazione, come il colore, un timer, ecc.

```
# Dimensioni dello spazio di simulazione
size = vec2f(500, 300)

class Particle(object):
    '''Rappresenta una particella tramite i suoi'''
    def __init__(self):
        '''parametri di simulazione'''
        # Raggio
        self.radius = 25.0
        # Posizione, inizialmente al centro
        self.pos = size/2
        # Velocità, inizialmente 0
        self.vel = vec2f(0,0)
        # Accelerazione, inizialmente 0
        self.acc = vec2f(0,0)
        # Massa
        self.mass = 1.0
        # Se simulata calcoleremo il moto
        self.simulated = True
        # Colore
        self.color = QColor(128,128,128,200)
        # Colore quando non simulata
        self.color_paused = QColor(128,255,128,200)
        # Timer, se 0 disattivato
        self.timer = 0
        # Attiva le collisioni tra particelle
        self.collisions = False
```

## 19.4 STRUTTURA DEL PROGRAMMA

Il simulatore userà una lista globale `particles` per mantenere le particelle e il widget per la grafica interattiva usato nei capitoli precedenti. Le particelle sono inizializzate con la funzione `init()`, che per ora ne crea solo una immobile. Per il momento, la funzione `paint()` calcola il nuovo stato del sistema con `update()` e poi disegna le particelle con `draw()`. In questa funzione, ogni particella è disegnata come un cerchio con una linea dal centro alla direzione della velocità.

```
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from gwidget import run_app

# Lista delle particelle
particles = []

def init():
    '''Crea e inizializza le particelle,
    per adesso una sola particella immobile'''
    global particles
    particles = [Particle()]

def update():
    '''Aggiorna posizioni e moto delle particelle'''
    pass

def v2qt(v):
    '''Converte un vec2f v in un vettore di Qt'''
    return QPointF(v.x, v.y)

def clear(painter):
    '''Pulisce lo schermo'''
    painter.fillRect(0,0,size.x,size.y,
        QColor(255,255,255))
```

```

def set_color(painter,c):
    '''Setta il color di brush e pen'''
    painter.setPen(QPen(QColor(0,0,0,c.alpha()),2))
    painter.setBrush(c)

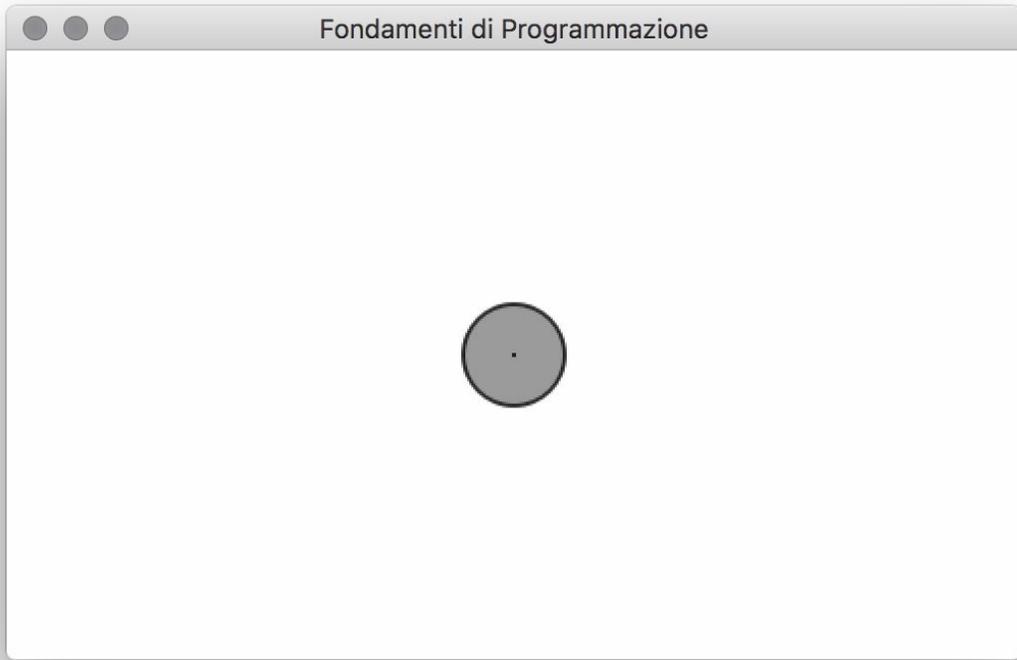
def draw(painter):
    '''Disegna le particelle'''
    clear(painter)
    for p in particles:
        if p.simulated:
            set_color(painter,p.color)
        else:
            set_color(painter,p.color_paused)
    painter.drawEllipse(v2qt(p.pos),
        p.radius, p.radius)
    # Disegna una linea dal centro della particella
    # che ne indica la velocità
    painter.drawLine(v2qt(p.pos),
        v2qt(p.pos+normalize(p.vel)*p.radius))

def paint(painter):
    '''Aggiorna la simulazione ad ogni frame'''
    # Aggiorna posizioni e moto delle particelle
    update()
    # Disegna il nuovo stato della simulazione
    draw(painter)

# Inizializza le particelle
init()

# Esegue la simulazione chiamando paint ogni frame
run_app(paint, int(size.x), int(size.y))

```



## 19.5 SIMULAZIONE DEL MOTO

Il moto delle particelle è guidato dalle equazioni viste sopra che implementiamo nella funzione `update()`. Se la particella si muove, uscirà dallo spazio di simulazione. Perciò aggiungiamo il rimbalzo sui bordi dello spazio controllando, per ogni bordo, se la particella l'ha superato. Se è accaduto, riportiamo la particella sul bordo e dirigiamo la velocità nel verso opposto. Introduciamo una funzione per creare e inizializzare una particella e modifichiamo quindi anche la funzione `init()`.

```
def handle_walls():
    '''Gestisce le collisioni particelle-bordi'''
    for p in particles:
        # Bordo sinistro
        if p.pos.x < p.radius:
            p.pos.x = p.radius
            p.vel.x = abs(p.vel.x)
        # Bordo destro
        if p.pos.x > size.x - p.radius:
            p.pos.x = size.x - p.radius
            p.vel.x = -abs(p.vel.x)
        # Bordo alto
        if p.pos.y < p.radius:
            p.pos.y = p.radius
            p.vel.y = abs(p.vel.y)
        # Bordo basso
        if p.pos.y > size.y - p.radius:
            p.pos.y = size.y - p.radius
            p.vel.y = -abs(p.vel.y)

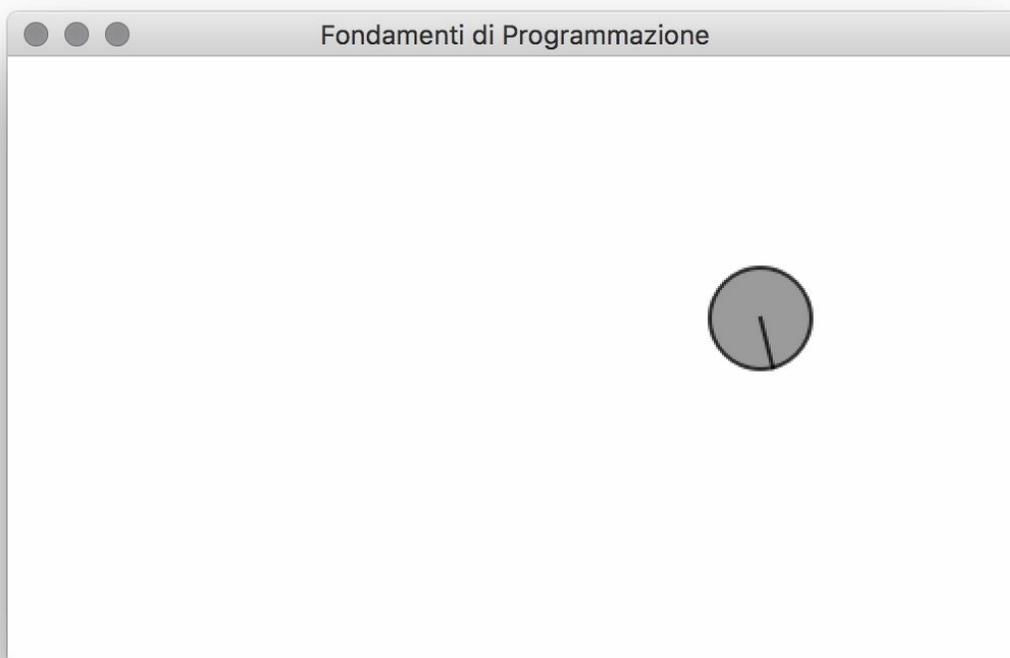
def update():
    '''Aggiorna posizioni e moto delle particelle'''
    for p in particles:
        # salta se non simulata
        if not p.simulated: continue
        # aggiorna posizione con la velocità
        p.pos += p.vel
```

```
# Gestisce le collisioni particelle-bordi
handle_walls()

def init_particle():
    '''Crea e inizializza una particella'''
    p = Particle()
    p.vel = vec2f(1,4)
    return p

def init():
    '''Crea e inizializza le particelle'''
    global particles
    particles = [init_particle()]

init()
run_app(paint, int(size.x), int(size.y))
```



## 19.6 PARAMETRI DI SIMULAZIONE

Per convenienza, introduciamo una classe `Params` che raccoglie tutti i parametri della simulazione, come ad esempio il numero di particelle da creare, la velocità massima, la densità, le forze attive, la dissolvenza, se la posizione è relativa al mouse e un timer. I parametri della simulazione corrente sono memorizzati nella variabile globale `params` di tipo `Params`.

```
class Params:
    '''Parametri della simulazione'''
    def __init__(self):
        # Numero particelle
        self.num = 4
        # Minimo e massimo raggio
        self.radius_min = 25.0
        self.radius_max = 25.0
        # Massima intensità delle velocità
        self.vel = 4.0
        # Densità
        self.density = 1.0/(pi*25*25)
        # Fattore d'attrazione verso il mouse
        self.force_mouse = 0.0
        # Fattore di decelerazione
        self.force_drag = 0.002
        # Max intensità forza casuale
        self.force_random = 0.0
        # Accelerazione di gravità
        self.force_gravity = 0.0
        # Dissolvenza dei movimenti precedenti
        self.fade = True
        # Posizione relativa al mouse
        self.mouse = False
        # Max timer
        self.timer = 0

# Parametri della simulazione
params = Params()
```

## 19.7 CREAZIONE DI PARTICELLE

Modifichiamo la funzione `init_particle()` per creare una particella con raggio, posizione e velocità casuali, ma nei limiti imposti dai parametri definiti in precedenza. Infine una piccola modifica alla funzione `init()` per creare un numero di particelle desiderato. Infine, modifichiamo la funzione `clear()` per il disegno tenendo conto dell'eventuale dissolvenza dei movimenti precedenti.

```
def init_particle():
    '''Crea e inizializza una particella'''
    p = Particle()
    # Raggio random
    p.radius = uniform(params.radius_min,
                       params.radius_max)
    # Posizione random
    p.pos = random_pos(0, 0, size.x, size.y)
    # Velocità random ma limitata
    p.vel = random_vec(params.vel)
    # Massa = area x densità
    p.mass = pi*(p.radius**2)*params.density
    return p

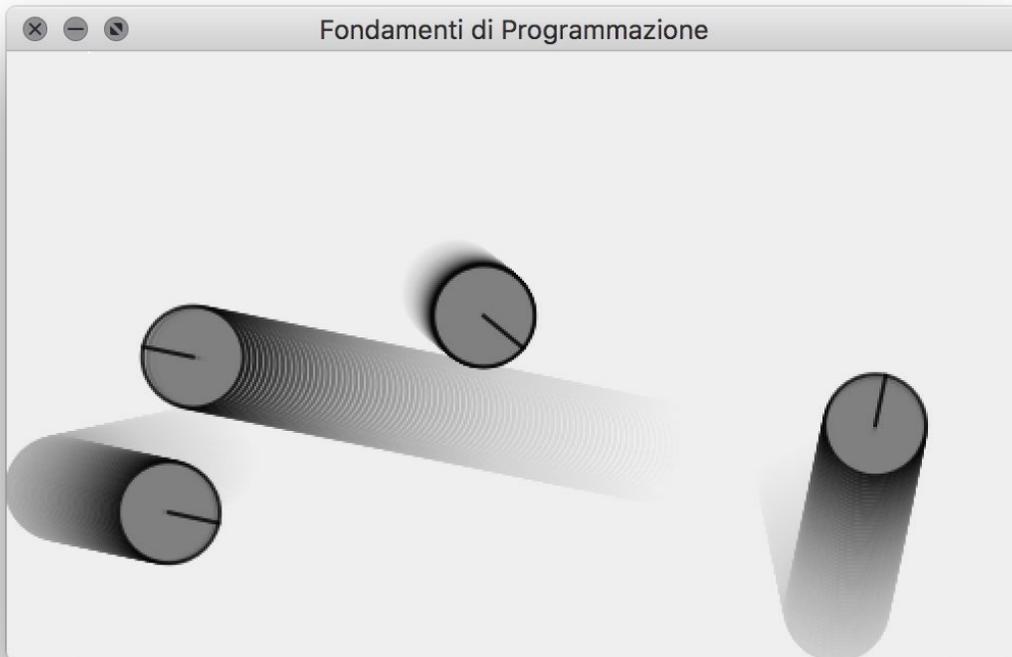
def init():
    '''Crea e inizializza le particelle'''
    global particles
    particles = []
    # Ora ci sono più particelle
    for _ in range(params.num):
        particles += [init_particle()]

def clear(painter):
    '''Pulisce lo schermo'''
    if params.fade:
        fade = 8
    else:
        fade = 255
    painter.fillRect(0,0,size.x,size.y,
```

```
QColor(255,255,255, fade))
```

```
init()
```

```
run_app(paint, int(size.x), int(size.y))
```



## 19.8 FORZE

Ora aggiungiamo varie forze, i cui parametri sono memorizzati in `Params`. Considereremo una forza casuale di lunghezza massima `force_random`, l'accelerazione di gravità `force_gravity`, una forza che rallenta la particella in modo proporzionale alla velocità `force_drag`, e una forza che attrae la particella nella direzione del mouse `force_mouse`. Per applicare le forze modifichiamo `update()`, passandogli in input la posizione del mouse dalla funzione `paint()`.

```
def handle_forces(mouse_pos):
    '''Aggiorna l'accelerazione di ogni particella'''
    for p in particles:
        # salta se non simulata
        if not p.simulated: continue
        # Parte da accelerazione 0
        p.acc = vec2f(0,0)
        # Accelerazione proporzionale alla distanza
        # dal mouse
        p.acc += (normalize(mouse_pos - p.pos) *
                 params.force_mouse)
        # Accelerazione casuale
        p.acc += (random_vec(params.force_random) /
                 p.mass)
        # Decelerazione proporzionale alla velocità
        p.acc += -p.vel*params.force_drag / p.mass
        # Accelerazione di gravità
        p.acc += vec2f(0, params.force_gravity)

def update(mouse_pos):
    '''Aggiorna posizioni e moto delle particelle'''
    # Aggiorna l'accelerazione di ogni particella
    handle_forces(mouse_pos)
    # Aggiorna velocità e posizione
    for p in particles:
        if not p.simulated: continue
        p.vel += p.acc
```

```

    p.pos += p.vel
    # Gestisce le collisioni particelle-bordi
    handle_walls()

def paint(painter):
    '''Aggiorna la simulazione ad ogni frame'''
    # Aggiorna posizioni e moto delle particelle,
    # tenendo anche conto del mouse
    update(vec2f(painter.info.mouse_x,
                painter.info.mouse_y))
    draw(painter)

```

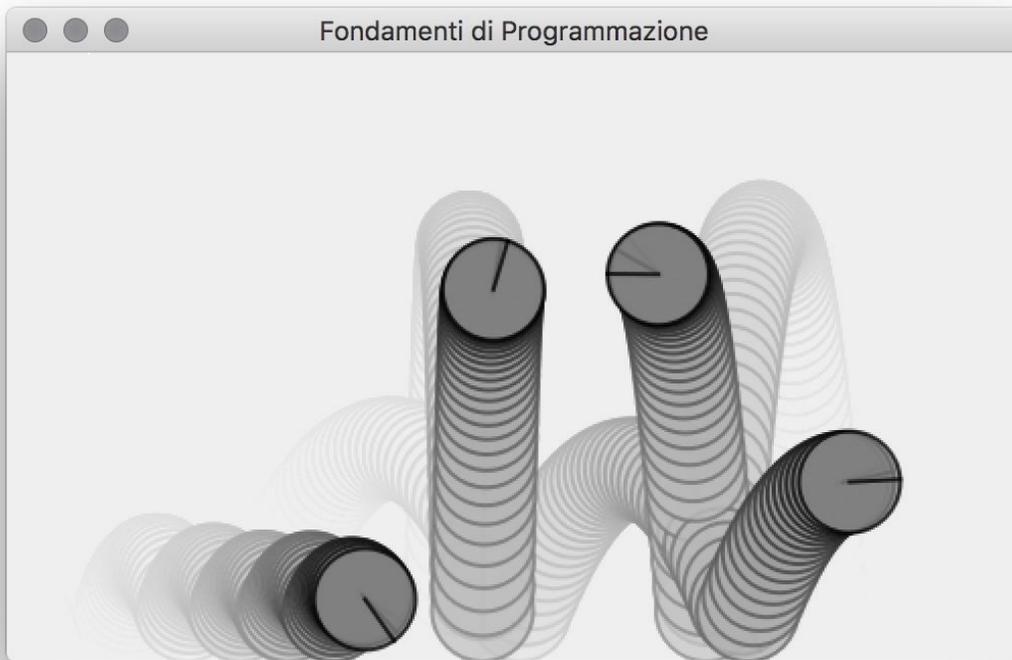
Possiamo ora variare la nostra simulazione cambiando semplicemente i parametri.

```

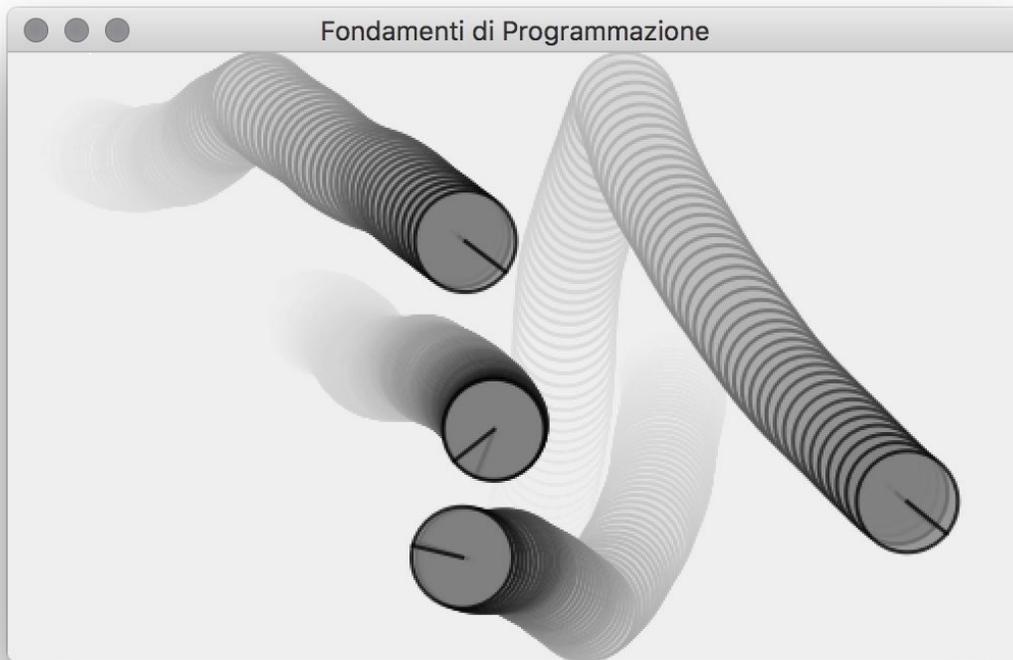
params = Params()
params.force_gravity = 0.5

init()
run_app(paint, int(size.x), int(size.y))

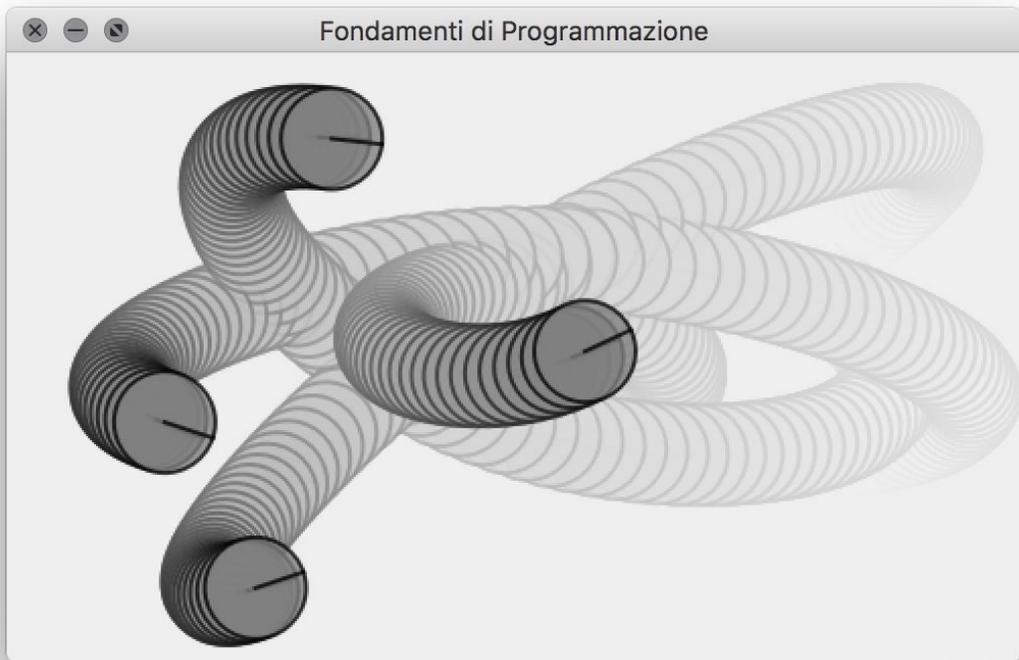
```



```
params = Params()  
params.force_random = 0.5  
  
init()  
run_app(paint, int(size.x), int(size.y))
```



```
params = Params()  
params.force_mouse = 0.5  
  
init()  
run_app(paint, int(size.x), int(size.y))
```



## 19.9 INTERAZIONE

Permettiamo all'utente di interagire tramite il mouse con le particelle. L'utente seleziona una particella cliccando con il mouse su di essa. Finché il mouse rimane premuto, la particella segue la posizione del mouse. Quando il mouse è rilasciato, la particella torna ad essere simulata con la velocità impartita dalle ultime posizioni del mouse. Questo permette di "lanciare" la particella tramite mouse.

Per implementare questo comportamento, memorizziamo la particella selezionata nella variabile globale `grabbed`. Quando l'utente clicca sullo schermo, una particella è selezionata se la distanza tra la posizione del mouse e il centro della particella è inferiore al suo raggio. In questo caso, modifichiamo `grabbed` e impostiamo l'attributo `simulated` a `False`, cosicché il movimento della particella non sia più gestito dalla simulazione ma dipenda solamente dal movimento del mouse fintanto che il mouse rimane premuto. Da questo momento, la posizione della particella selezionata è quella del mouse finché il mouse rimane premuto. Al rilascio del mouse, impostiamo la velocità della particella in modo proporzionale al vettore dalla posizione precedente del mouse a quella attuale, e poi rimettiamo la particella in simulazione riportando il suo attributo `simulated` a `True`.

```
# Eventuale particella presa con il mouse
grabbed = None

def grab(mouse_pressed, mouse_pos, mouse_lpos):
    '''Gestisce la particella selezionata'''
    global grabbed
    # Se il mouse è premuto
    if mouse_pressed:
        # Se non c'è una particella presa
        if not grabbed:
            # Controlla per ogni particella se
            for p in particles:
                # Se è vicina al mouse
                if length(p.pos-mouse_pos)<p.radius:
```

```

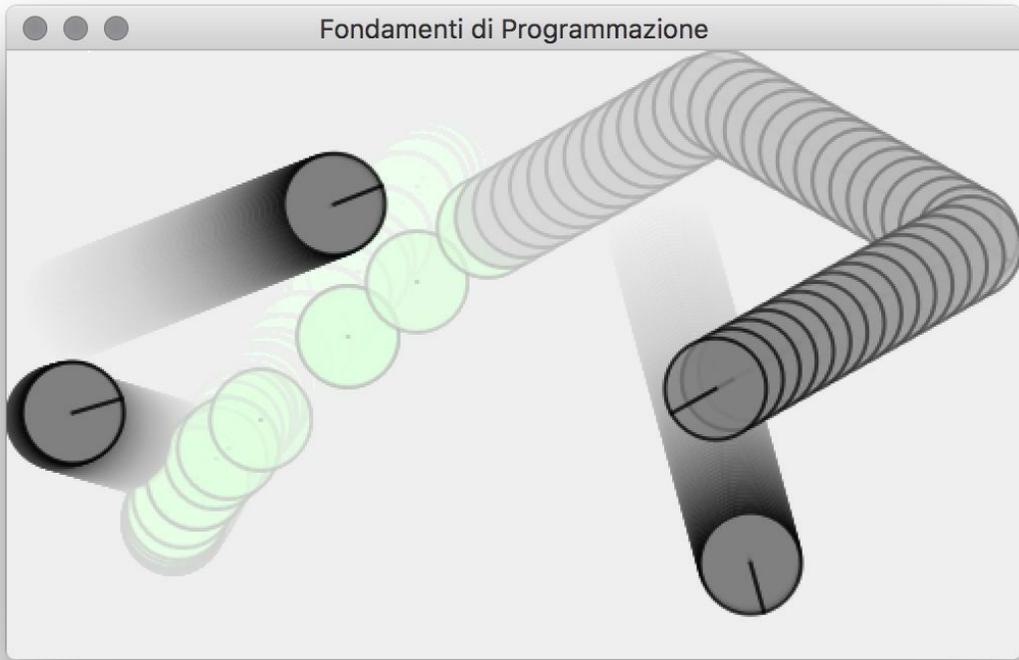
        # prendila
        grabbed = p
        # e escludila dalla simulazione
        grabbed.simulated = False
        grabbed.vel = vec2f(0,0)
        break
    # Se c'è una particella presa,
    else:
        # spostala con il mouse
        grabbed.pos += mouse_pos - mouse_lpos
# Il mouse non è premuto e una particella è presa
elif grabbed:
    # Rimettila in simulazione lanciandola con
    # velocità dipendente dal mouse
    grabbed.simulated = True
    v = mouse_pos - mouse_lpos
    grabbed.vel = clamp(v, 16)
    grabbed = None

def paint(painter):
    '''Aggiorna la simulazione ad ogni frame'''
    # Gestisci la presa di una particella con il mouse
    grab(painter.info.mouse_pressed,
        vec2f(painter.info.mouse_x,
            painter.info.mouse_y),
        vec2f(painter.info.mouse_px,
            painter.info.mouse_py))
    update(vec2f(painter.info.mouse_x,
        painter.info.mouse_y))
    draw(painter)

params = Params()

init()
run_app(paint, int(size.x), int(size.y))

```



## 19.10 GENERAZIONE CONTINUA

Finora le particelle sono inizializzate quando create e i loro parametri rimangono invariati durante l'intera simulazione. Per creare effetti come fuochi d'artificio, possiamo inizializzare le particelle con un timer che, fungendo da conto alla rovescia, rappresenta il numero di frame per cui la particella manterrà i valori dei suoi parametri attuali. Nella funzione `update()`, ad ogni frame il timer di ogni particella temporizzata è decrementato e quando raggiunge lo zero, la particella è inizializzata con nuovi parametri. Utilizziamo i parametri `params.timer` per valore massimo dei timer delle particelle e `params.mouse` che determina se la posizione di una particella è inizializzata con quella del mouse o è generata in modo casuale. Inoltre aumentiamo il numero di particelle e diminuiamo i loro raggi. Per ogni particella, utilizziamo l'attributo `timer`, il cui valore `0` significa che la particella non è temporizzata, cioè la sua inizializzazione non è regolata dal timer.

```
params = Params()  
  
params.num = 120  
params.radius_min = 5.0  
params.radius_max = 5.0  
params.force_gravity = 0.1  
params.fade = False  
params.mouse = True  
params.timer = 120  
params.Fade = False
```

Modifichiamo ora la funzione `init_particle()` per far sì che possa essere chiamata anche per inizializzare una particella già creata. La funzione prenderà ora in input anche la posizione del mouse, che all'inizio è l'origine. Inoltre modifichiamo la funzione `update()` per gestire le particelle temporizzate.

```

def init_particle(mouse_pos=vec2f(0,0), p=None):
    '''Crea e inizializza una particella, o inizializza
    una particella già esistente, inizializza la
    posizione con quella del mouse o random e
    imposta un timer'''
    # Reusa la particella se già esistente
    if not p: p = Particle()
    # Timer
    p.timer = randint(params.timer/2, params.timer)
    p.radius = uniform(params.radius_min,
        params.radius_max)
    # Posizione relativa al mouse o casuale
    if params.mouse:
        p.pos = vec2f(mouse_pos.x, mouse_pos.y)
    else:
        p.pos = random_pos(0,0,size.x,size.y)
    p.vel = random_vec(params.vel)
    p.mass = pi*(p.radius**2)*params.density
    return p

def handle_timers(mouse_pos):
    '''Gestisce il timer delle particelle'''
    for p in particles:
        if not p.simulated: continue
        # Se non temporizzata, ignorala
        if not p.timer: continue
        # decrementa il timer
        p.timer -= 1
        # e se e' arrivato a zero
        # reinizializza la particella
        if p.timer < 1:
            init_particle(mouse_pos, p)

def update(mouse_pos):
    '''Aggiorna posizioni e moto delle particelle'''
    # Gestisce il timer delle particelle
    handle_timers(mouse_pos)
    handle_forces(mouse_pos)
    for p in particles:
        if not p.simulated: continue
        p.vel += p.acc
        p.pos += p.vel

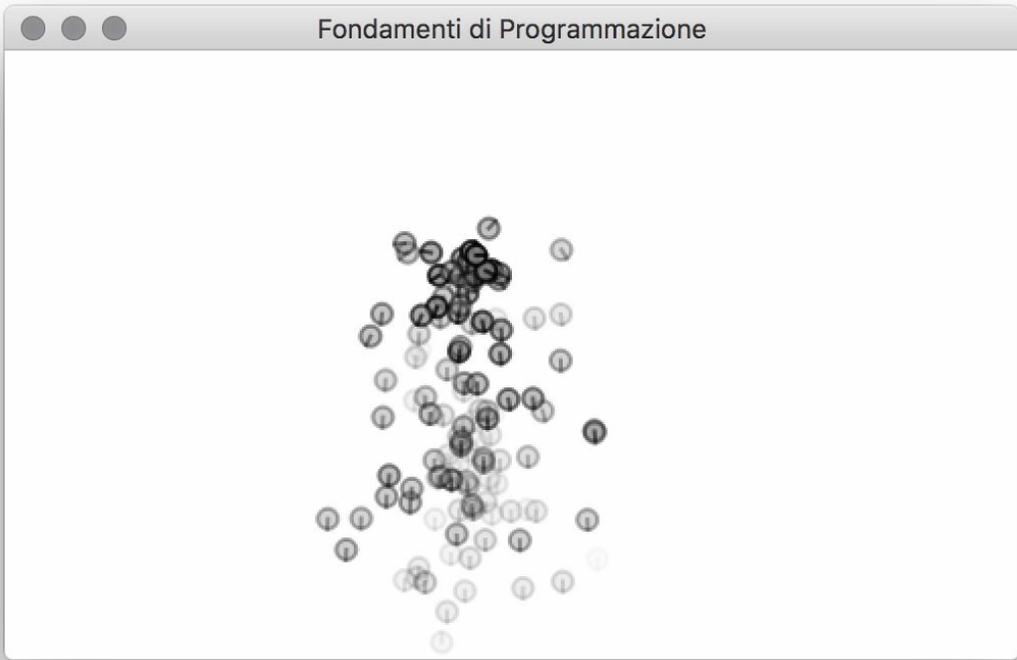
```

```
handle_walls()
```

Per far sì che l'inizializzazione continua delle particelle non risulti un effetto troppo brusco, rendiamo la trasparenza delle particelle temporizzate proporzionale al timer. Così più il timer si avvicina a zero e più la particella diventa trasparente.

```
def draw(painter):
    '''Disegna le particelle'''
    clear(painter)
    for p in particles:
        # La trasparenza dipende dal timer
        if p.timer:
            set_color(painter, QColor(p.color.red(),
                                      p.color.green(), p.color.blue(),
                                      p.timer*2))
        elif p.simulated:
            set_color(painter,p.color)
        else:
            set_color(painter,p.color_paused)
    painter.drawEllipse(v2qt(p.pos),
                        p.radius, p.radius)
    painter.drawLine(v2qt(p.pos),
                    v2qt(p.pos+normalize(p.vel)*p.radius))

init()
run_app(paint, int(size.x), int(size.y))
```



## 19.11 COLLISIONI

Fino ad ora, le particelle non si scontrano tra loro. Gestire le collisioni tra particelle è molto più complesso delle collisioni con i bordi. La ragione è che la fisica dell'impatto richiede la decomposizione delle velocità proiettandole sull'asse che passa per i centri delle particelle e sull'asse ad esso perpendicolare e richiedendo anche di tener conto delle masse delle particelle. Riportiamo ora il codice per le collisione che include commenti sull'implementazione. Con questo programma si può giocare al biliardo con le particelle.

```
def handle_collisions():
    '''Gestisce le collisioni tra particelle'''
    if not params.collisions:
        return
    for p in particles:
        for p1 in particles:
            if p1 is p: continue
            # Vettore tra i centri delle particelle
            pp = normalize(p1.pos - p.pos)
            # Distanza dei centri delle particelle
            d = length(p1.pos - p.pos)
            # Somma dei raggi
            rr = p1.radius + p.radius
            # Se la distanza è maggiore dei raggi
            if d >= rr:
                # non c'è collisione
                continue
            # Se la seconda particella è simulata
            if p1.simulated:
                # Punto medio dei centri
                m = (p.pos + p1.pos)/2
                # Posiziona le particelle in modo
                p.pos = m - pp * rr/2
                # che siano tangenti
                p1.pos = m + pp * rr/2
            # Determina le velocità
```

```

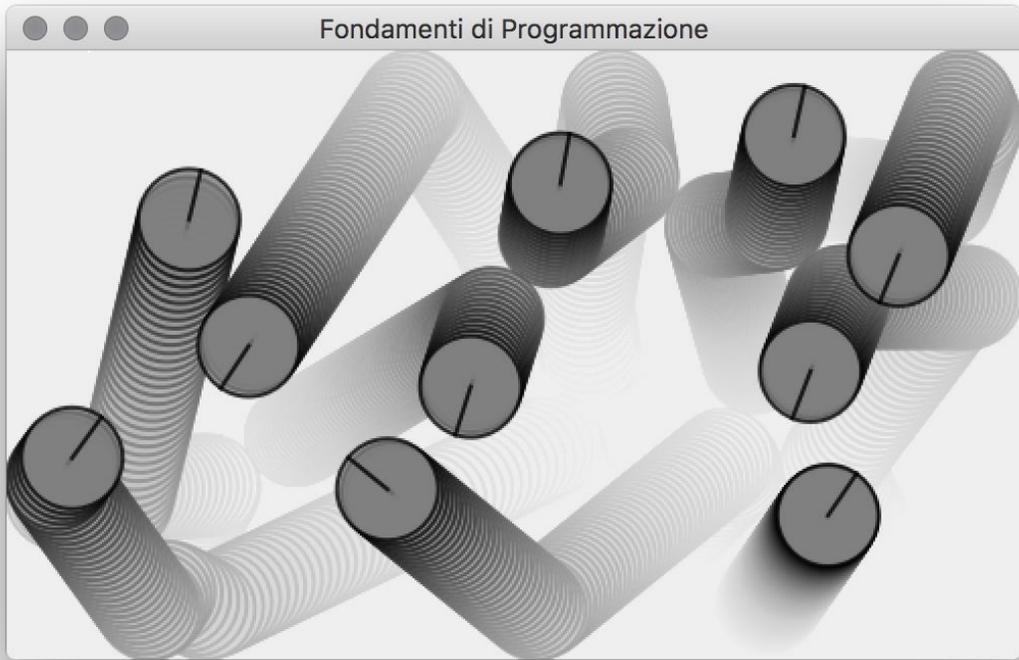
    vo0 = pp*dot(p.vel, pp)
    vp0 = p.vel - vo0
    vo1 = pp*dot(p1.vel, pp)
    vp1 = p1.vel - vo1
    if dot(vo1 - vo0, pp) < 0:
        m = p1.mass+p.mass
        vn0 = ( vo0 * (p.mass-p1.mass) +
                vo1*2*p1.mass) / m
        vn1 = ( vo1 * (p1.mass-p.mass) +
                vo0*2*p.mass) / m
        p.vel = vn0 + vp0
        p1.vel = vn1 + vp1
    # Se la seconda particella non è simulata
    else:
        p.pos = p1.pos - pp * rr
        vo0 = pp * dot(p.vel, pp)
        vp0 = p.vel - vo0
        if dot(-vo0, pp) < 0:
            p.vel = vp0 - vo0

def update(mouse_pos):
    '''Aggiorna posizioni e moto delle particelle'''
    # Gestisce il timer delle particelle
    handle_timers(mouse_pos)
    # Aggiorna l'accelerazione di ogni particella
    handle_forces(mouse_pos)
    # Aggiorna posizione e velocità
    for p in particles:
        if not p.simulated: continue
        p.vel += p.acc
        p.pos += p.vel
    handle_walls()
    # Gestisce le collisioni tra particelle
    handle_collisions()

params = Params()
params.collisions = True
params.num = 10

init()
run_app(paint, int(size.x), int(size.y))

```



## 20 APPLICAZIONI WEB E CLI

Fino ad ora abbiamo visto applicazioni interattive con interfacce grafiche. In questo capitolo introdurremo le basi per la applicazioni Web e da terminale implementando queste interfacce sulla stessa applicazione.

## 20.1 STRUTTURA DELL'APPLICAZIONE

Come esempio sviluppiamo un'applicazione per la gestione di una lista di promemoria, o *todo list*, memorizzata in un file di testo, con un promemoria per ogni riga del file. Ogni promemoria è costituito dal simbolo `-`, che indica l'inizio riga, il simbolo `[_]` o `[x]` per promemoria incompleti o no, un numero univoco che identifica il promemoria formattato con `@xxxx`, ed infine il titolo del promemoria, fino a fine riga. Ad esempio,

```
- [_] @0000 scrivere libro  
- [x] @0001 mangiare un gelato buono
```

Questo semplice formato ci permette di agire sui promemoria direttamente editando il testo con un qualunque editor, ed allo stesso tempo di creare interfacce specifiche. Nella nostra applicazione l'utente non dovrà preoccuparsi di salvare o caricare esplicitamente i dati, come avviene comunemente nelle applicazioni Web o smartphone. I comandi che la nostra applicazione dovrà supportare sono la stampa dell'elenco dei promemoria, la creazione di nuovi promemoria, la spuntatura del flag di completamento, la cancellazione di un promemoria e lo spostamento di un elemento in alto o in basso nella lista. Ad ogni comando, il file di testo sarà cambiato opportunamente.

## 20.2 FUNZIONALITÀ LOGICHE DELL'APPLICAZIONE

Scriviamo ora una serie di funzioni che implementa le funzionalità logiche dell'applicazione, senza preoccuparci di come verranno chiamate queste funzioni dalle interfacce. Tutte le funzioni agiranno sul file al percorso `filename` usando le funzioni `load()` e `save()` che manipolano il formato. L'utente può configurare il file dell'applicazione con `init()`. `next_uid()` crea un nuovo identificativo, nel nostro caso il numero successivo al maggiore dei numeri usati come id, e `find()` trova l'indice della riga per l'identificativo specificato.

```
import os

# percorso del file todo
filename = 'todo.txt'

def load():
    '''Carica la lista dei promemoria e ritorna la
    lista delle linee del file'''
    with open(filename) as f:
        return f.read().splitlines()

def save(lst):
    '''Salva la lista di promemoria come testo'''
    with open(filename, 'w') as f:
        return f.write('\n'.join(lst))

def init(clear=True):
    '''Inizializza il file dei promemoria'''
    if clear:
        save([])
    else:
        if os.path.exists(filename): return
        save('')

def next_uid(lst):
    '''Calcola un nuovo identificativo univoco'''
```

```

max_uid = 0
for line in lst:
    uid = line.split()[2]
    num = int(uid[1:])
    if num > max_uid:
        max_uid = num
return '@{:04}'.format(max_uid+1)

def find(lst, uid):
    '''Ritorna l'indice di riga del promemoria
    con identificativo uid'''
    for i, line in enumerate(lst):
        if uid in line.lower().split():
            return i
    return -1

```

Implementiamo ora le funzionalità dell'applicazione. La funzione `add(title)` aggiunge un promemoria dal titolo `title` alla fine della lista. La funzione `print_txt(searched,search)` stampa i promemoria che contengono la stringa `filter`, che è opzionale, includendo i promemoria completati se `checked` è `True`. La funzione `check(uid,checked)` cambia lo stato del promemoria all'identificativo `uid`. Tutte le funzioni ritorneranno vero o falso per esecuzioni corrette o con errori.

```

def add(title,uid=None):
    '''Aggiunge un promemoria alla lista'''
    lst = load()
    if not uid:
        uid = next_uid(lst)
    lst += ['- [_] {} {} \n'.format(uid,title.strip())]
    save(lst)
    return True

def print_txt(checked=False,search=None):
    '''Stampa la lista. Se checked è falso, esclude
    i promemoria completati. Se search non è None,
    includi solo i promemoria che la contengono'''
    lst = load()
    for line in lst:
        if (not checked and
            '[x]' in line.lower()): continue

```

```

        if (search and (search.lower() not in
            line.lower().split())): continue
        print(line)
    return True

def check(uid,checked):
    '''Setta la stato del promemoria uid come
    spuntato o no'''
    lst = load()
    pos = find(lst,uid)
    if pos < 0: return False
    if checked:
        lst[pos] = lst[pos].replace('[_]','[x]')
    else:
        lst[pos] = lst[pos].replace('[x]','[_]')
    save(lst)
    return True

```

Per testare queste funzionalità, creiamo una piccola lista di promemoria eseguendo una sequenza di comandi dell'applicazione. Si noti come non accediamo mai ai dati in modo esplicito. Eseguiamo i tests solo quando il codice è eseguito dalla console interattiva di Python, per assicurare l'uso del programma in modo applicazione.

```

# inicializziamo il sistema
if __name__ == '__console__':
    init()

# aggiungiamo due todo
if __name__ == '__console__':
    add('scrivere un libro')
    add('mangiare un gelato buono')
    add('andare a pescare')
    print_txt()

# Out: True
# Out: True
# Out: True
# Out: - [_] @0001 scrivere un libro
# Out: - [_] @0002 mangiare un gelato buono
# Out: - [_] @0003 andare a pescare
# Out: True

```

```

# spunta una todo
if __name__ == '__console__':
    check('@0002',True)
    print_txt()
# Out: True
# Out: - [_] @0001 scrivere un libro
# Out: - [_] @0003 andare a pescare
# Out: True

# stampa
if __name__ == '__console__':
    print_txt(checked=True)
# Out: - [_] @0001 scrivere un libro
# Out: - [x] @0002 mangiare un gelato buono
# Out: - [_] @0003 andare a pescare
# Out: True

if __name__ == '__console__':
    check('@0002',False)
    print_txt(search='pescare')
# Out: True
# Out: - [_] @0003 andare a pescare
# Out: True

if __name__ == '__console__':
    print_txt(search='@0001')
# Out: - [_] @0001 scrivere un libro
# Out: True

```

Implementiamo infine le funzioni `up(uid)` e `down(uid)` che spostano un elemento in su o giù di uno, e la funzione `erase(uid)` che cancella l'elemento `uid`.

```

def up(uid):
    '''Muove in sù un promemoria'''
    lst = load()
    pos = find(lst,uid)
    if pos < 0: return False
    if pos == 0: return False
    lst[pos], lst[pos-1] = lst[pos-1], lst[pos]

```

```

save(lst)
return True

def down(uid):
    '''Muove in giù un promemoria'''
    lst = load()
    pos = find(lst,uid)
    if pos < 0: return False
    if pos == len(lst)-1: return False
    lst[pos], lst[pos+1] = lst[pos+1], lst[pos]
    save(lst)
    return True

def erase(uid):
    '''Cancella un promemoria'''
    lst = load()
    pos = find(lst,uid)
    if pos < 0: return False
    lst = lst[:pos] + lst[pos+1:]
    save(lst)
    return True

# facciamo alcuni tests
if __name__ == '__console__':
    print_txt()
# Out: - [_] @0001 scrivere un libro
# Out: - [_] @0002 mangiare un gelato buono
# Out: - [_] @0003 andare a pescare
# Out: True

if __name__ == '__console__':
    up('@0002')
    print_txt()
# Out: True
# Out: - [_] @0002 mangiare un gelato buono
# Out: - [_] @0001 scrivere un libro
# Out: - [_] @0003 andare a pescare
# Out: True

if __name__ == '__console__':
    erase('@0001')
    print_txt()
# Out: True

```

```
# Out: - [_] @0002 mangiare un gelato buono  
# Out: - [_] @0003 andare a pescare  
# Out: True
```

```
if __name__ == '__console__':  
    down('@0002')  
    print_txt()
```

```
# Out: True  
# Out: - [_] @0003 andare a pescare  
# Out: - [_] @0002 mangiare un gelato buono  
# Out: True
```

## 20.3 DECORATORI

In Python, i decorator sono una sintassi speciale per modificare il comportamento di una funzione o aggiungere informazioni addizionali alla stessa. I decorator sono un meccanismo molto flessibile che noi useremo solo in congiunzione con librerie per creare interfacce. I decorator si specificano con un lista di dichiarazioni prima di una funzione. Ogni dichiarazione ha la forma `@decorator(params)` dove `decorator` è il nome del decoratore e `params` la lista dei suoi parametri.

## 20.4 INTERFACCIA A RIGA DI COMANDO

Abbiamo già usato un programma che ha un'interfaccia a riga di comando, `python3`, che prende come primo argomento opzionale il nome di un file, ad esempio `python3 file.py`. Specifichiamo ora una riga di comando per la nostra applicazione `todo.py`.

```
todo.py print [-s search] [-c]
    # stampa la lista di promemria
    # opzione -s: filtra con search
    # opziona -c: include spuntati
todo.py add "title" [-u uid]
    # aggiunge il promemoria title
    # opzione -u: assegna uid
todo.py erase uid
    # cancella il promemoria uid
todo.py up uid
    # muove uid in alto
todo.py down uid
    # muove uid in basso
```

Come si può vedere questi comandi sono già implementati nella varie funzioni di cui sopra. Basta solo chiamare queste funzioni in modo appropriato leggendo i parametri dalla riga di comando del terminale. Per farlo, potremmo gestire manualmente la conversione dei comandi, cosa che risulta tediosa e prona ad errori.

Per scrivere la nostra interfaccia a riga di comando useremo invece la libreria **Click** installabile con `pip3 install click`. Questa libreria definisce un gruppo di decorator per creare applicazioni a riga di comando. L'idea di base è che ogni comando è una funzione Python che viene eseguita con argomenti copiati dalla riga di comando. Per mappare comandi a funzioni, usiamo il decoratore `@click.command(nome)` che espone la funzione successiva come comando `nome`. Per ogni argomento necessario usiamo `@click.argument(nome, required=True)`. Per ogni argomento opzionale,

usiamo `@click.option(nome)` dove il nome inizia con `--`. Per ulteriori informazioni su Click consultare la documentazione della libreria.

```
import click

@click.group()
def cli():
    '''Funzione vuota che raggruppa i comandi'''
    pass

# definisce il comando: add title [-u uid]
@cli.command('add')
@click.argument('title', required=True)
@click.option('-u', '--uid', default=None)
def add_cli(title, uid=None):
    '''interfaccia cli per add'''
    if add(title, uid): print('add new')
    else: print('add error')

# definisce il comando: print [-s search] [-c]
@cli.command('print')
@click.option('-s', '--search', default=None)
@click.option('-c', '--checked', is_flag=True,
              default=False)
def print_cli(search=None, checked=False):
    '''interfaccia cli per print'''
    print_txt(search=search, checked=checked)

# definisce il comando: check uid
@cli.command('check')
@click.argument('uid', required=True)
def check_cli(uid):
    '''interfaccia cli per check'''
    if check(uid, True): print('check', uid)
    else: print('check error')

# definisce il comando: uncheck uid
@cli.command('uncheck')
@click.argument('uid', required=True)
def uncheck_cli(uid):
    '''interfaccia cli per check'''
    if check(uid, False): print('uncheck', uid)
```

```

    else: print('uncheck error')

# definisce il comando: up uid
@cli.command('up')
@click.argument('uid', required=True)
def up_cli(uid):
    '''interfaccia cli per up'''
    if up(uid): print('up',uid)
    else: print('up error')

# definisce il comando: down uid
@cli.command('down')
@click.argument('uid', required=True)
def down_cli(uid):
    '''interfaccia cli per down'''
    if down(uid): print('down',uid)
    else: print('down error')

# definisce il comando: erase uid
@cli.command('erase')
@click.argument('uid', required=True)
def erase_cli(uid):
    '''interfaccia cli per erase'''
    if erase(uid): print('erase',uid)
    else: print('erase error')

# comando per lanciare l'applicazione web
# definito successivamente
@cli.command('web')
def web_cli():
    '''lancia l'applicazione web'''
    global run_web
    run_web = True

# chiama click se stiamo eseguendo come todo.py
if __name__ == '__main__':
    run_web = False
    cli(standalone_mode=False)

```

Con queste dichiarazioni la nostra applicazione è completa. Possiamo ora testarla eseguendola come un programma da riga di comando. Per farlo, basta salvare il codice di questo capitolo in `todo.py` e usare `python3`

`todo.py` su MacOS e Linux o `python3.exe todo.py` su Windows. Eseguendo l'applicazione senza comandi si ottiene un messaggio di errore, generato automaticamente, con informazioni su come usare l'applicazione. Aggiungendo comandi, possiamo vedere come usare l'applicazione in modo interattivo.

```
Terminal
[$ python3 todo.py
Usage: todo.py [OPTIONS] COMMAND [ARGS]...

Options:
  --help Show this message and exit.

Commands:
  add
  check
  down
  erase
  print
  uncheck
  up
$
```

```
Terminal
[$ python3 todo.py print
- [_] @0003 andare a pescare
- [_] @0002 mangiare un gelato buono
[$ python3 todo.py add 'andare a fare la spesa'
add new
[$ python3 todo.py check @0002
check @0002
[$ python3 todo.py print
- [_] @0003 andare a pescare
- [_] @0005 andare a fare la spesa
[$ python3 todo.py print -c
- [_] @0003 andare a pescare
- [x] @0002 mangiare un gelato buono
- [x] @0004 Un promemoria di prova
- [_] @0005 andare a fare la spesa
$
```

## 20.5 INTERFACCIA WEB

Per creare un'applicazione Web useremo la libreria `bottle.py` che è scaricabile gratuitamente o installabile con `pip3 install bottle`. L'applicazione Web visualizza una pagina HTML che include una barra per aggiungere un nuovo promemoria ed una per iniziare una ricerca, seguite dalla lista dei promemoria. Per ogni promemoria includiamo quattro bottoni per chiamare le funzioni `check()`, `up()`, `down()` e `erase()` sul promemoria scelto.

Per chiamare le varie funzioni utilizzeremo un decoratore `@bottle.route(url)` che definisce la corrispondenza di un indirizzo Web con la funzione. In altre parole, se nella bara di navigazione del browser si immette `url` la funzione decorata viene chiamata e il suo risultato viene inviato al browser come HTML. Notiamo che `url` è relativo all'applicazione corrente. Nel nostro caso, utilizzeremo in server Web locale creato automaticamente da Bottle, chiamato `localhost`. Il decoratore di Bottle supporta anche la specifica delle parti dell'url che vengono mappate a parametri della funzioni. Nel nostro caso supporteremo i seguenti comandi:

```
/          applicazione base
/add       aggiungi un promemoria
/check/<uid> spunta un promemoria
/uncheck/<uid> spunta un promemoria
/up/<uid>   mouvi un promemoria
/down/<uid> mouvi un promemoria
/erase/<uid> cancella un promemoria
```

Dato che tutti i comandi necessitano di ritorna la pagine HTML al browser, scriviamo una funzione `print_html()` che ritorna la pagina corrente. Per questo utilizzeremo un template per la pagina intera, in cui poi inseriamo la lista dei promemoria. Per poter integrare gli elementi attivi dell'interfaccia usiamo il tag `<form>` che contiene elementi di tipo `<input type="tipo" formaction="url">`. Come tipo di elementi useremo `submit` per i checkbox,

`text` per il testo e `button` per i bottoni. Per ogni elemento specifichiamo l'azione da compiere dichiarando l'url da invocare con l'attributo `formaction`. Nel caso dei bottoni basta farlo direttamente nell'elemento. Per l'input di testo, inseriamo l'elemento, dichiarandone il nome con `name`, in una form separata che specificherà il proprio url con `action` e il metodo `method=get` che appena all'url il valore dell'elemento. Per la ricerca, manteniamo una variabile globale `web_search` che usiamo per filtrare i risultati.

```
web_search = ''

template_html = '''
<!DOCTYPE html>
<html lang="it">
  <head>
    <title>todo.py</title>
    <meta charset="utf-8">
  </head>
  <body>
    <form action="/add" method="get">
      <p><input type="text" name="title"
formaction="/add">Aggiungi</p>
    </form>
    <form action="/search" method="get">
      <p><input type="text" name="title" value="
{web_search}">Cerca</p>
    </form>
    <form>
      {items}
    </form>
  </body>
</html>
'''

template_item = '''
<p>
  <input type="submit" value={checked}
formaction="/{checkcmd}/{uid}">
  <input type="submit" value="U" formaction="/up/{uid}">
  <input type="submit" value="D" formaction="/down/{uid}">
  <input type="submit" value="X" formaction="/erase/{uid}">
  {title}
'''
```

```

</p>
...

def print_html():
    '''Ritorna la pagina HTML dell'applicazione'''
    lst = load()
    items = ''
    for line in lst:
        if (web_search and (web_search.lower()
            not in line.lower().split())): continue
        _, checked, uid, title = line.split(maxsplit=3)
        if checked == '[x]': checkcmd = 'uncheck'
        else: checkcmd = 'check'
        items += template_item.format(uid=uid,
            title=title, checked=checked,
            checkcmd=checkcmd)
    return template_html.format(items=items,
        web_search=web_search)

```

Possiamo ora connettere le varie funzioni agli URL predisposti dalla funzione `print_html()`. Bottle permette di definire sia URL statici, come `/`, che URL dinamici, come `/check/<uid>`. In questo secondo caso, Bottle accetta URL con qualsiasi valore per la parte `<uid>`. Questo valore verrà poi passato come stringa all'omonimo parametro della funzione. L'unica accortezza finale è di considerare la funzione `add()` che prende una stringa arbitraria in input. Bottle supporta questo mapping dando l'accesso ad un dizionario `request.query`. Utilizzeremo lo stesso modello per definire la ricerca.

```

import bottle

# comando add con titolo in request.query['title']
@bottle.route('/add')
def add_web():
    '''interfaccia web per add'''
    title = bottle.request.query.get('title')
    add(title)
    return print_html()

# setta la stringa di ricerca da request.query['title']
@bottle.route('/search')

```

```

def search_web():
    '''interfaccia web per settare il filtro
    di visualizzazione'''
    global web_search
    title = bottle.request.query.get('title')
    web_search = title
    return print_html()

# comando check uid
@bottle.route('/check/<uid>')
def check_web(uid):
    '''interfaccia web per check'''
    check(uid, True)
    return print_html()

# comando uncheck uid
@bottle.route('/uncheck/<uid>')
def uncheck_web(uid):
    '''interfaccia web per check'''
    check(uid, False)
    return print_html()

# comando up uid
@bottle.route('/up/<uid>')
def up_web(uid):
    '''interfaccia web per up'''
    up(uid)
    return print_html()

# comando down uid
@bottle.route('/down/<uid>')
def down_web(uid):
    '''interfaccia web per down'''
    down(uid)
    return print_html()

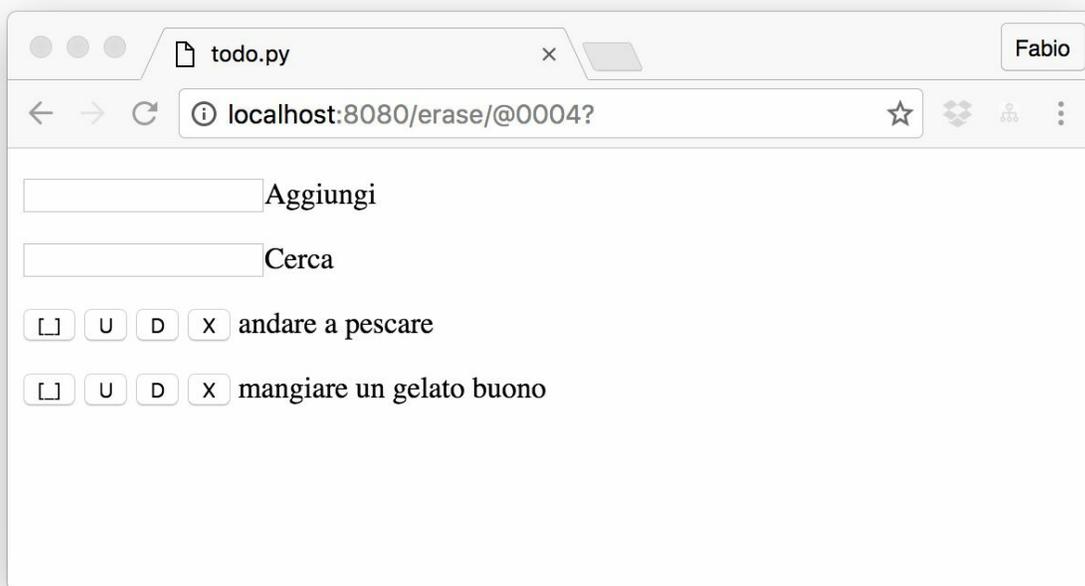
# comando erase uid
@bottle.route('/erase/<uid>')
def erase_web(uid):
    '''interfaccia web per erase'''
    erase(uid)
    return print_html()

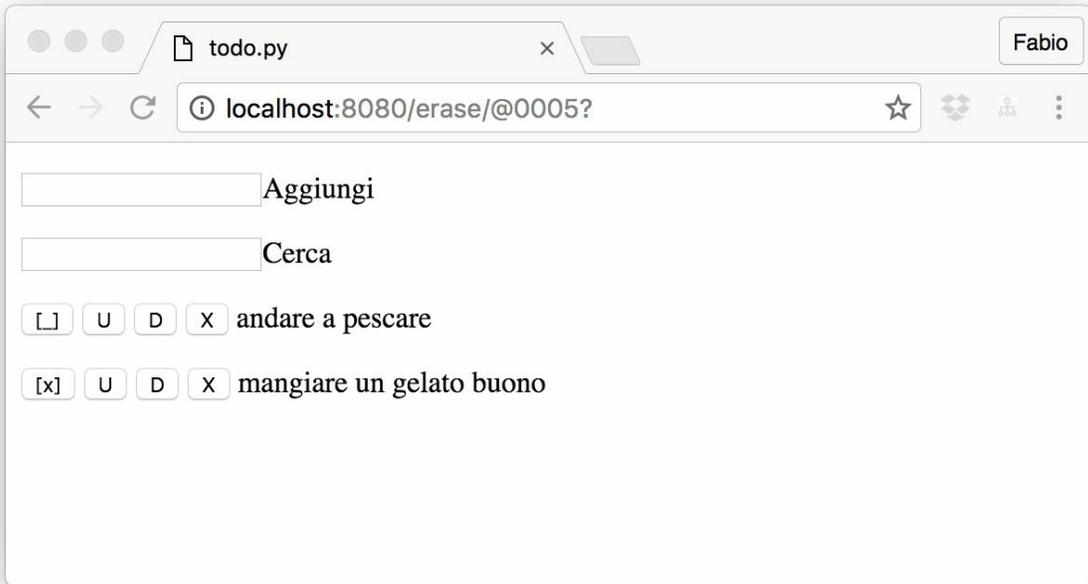
```

```
# pagina dell'applicazione
@bottle.route('/')
def print_web():
    '''interfaccia web per print_html'''
    return print_html()

if __name__ == '__main__' and run_web:
    bottle.run(debug=True)
```

Possiamo testare l'applicazione Web, lanciata con `python3 todo.py web`, considerando alcune operazioni a partire dallo stato iniziale usato sopra.



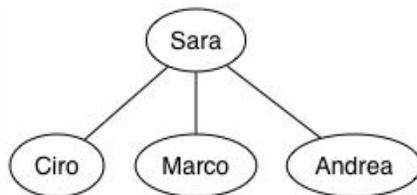


## 21 NAVIGARE LABIRINTI

Nei capitoli precedenti abbiamo visto che gli alberi permettono di rappresentare relazioni di contenimento tra elementi di vario tipo. Ma ci sono tanti scenari in cui le relazioni non sono gerarchiche e possono dar luogo a cicli. Ad esempio, le relazioni tra le città collegate da tratte aeree o le relazioni di amicizia tra le persone in una rete sociale. In questo capitolo, introdurremo i grafi, una struttura dati che permette di rappresentare relazioni più complesse.

## 21.1 GRAFI

Un *grafo* è un insieme di elementi che sono tra loro interconnessi da un qualche tipo di relazione. Nella terminologia dei grafi, gli elementi sono chiamati *nodi*, come per gli alberi, e le interconnessioni sono chiamate *archi*. Questa struttura si presta a rappresentare varie classi di problemi. Ad esempio, in una rete sociale i nodi sono le persone e gli archi sono le relazioni di amicizia. In una mappa, i nodi sono città e gli archi rappresentano strade di comunicazione. Esistono due principali tipi di grafi, *simmetrici* come nel caso delle amicizie, e *asimmetrici* come nel caso delle strade a senso unico. La figura che segue mostra un grafo molto semplice.



## 21.2 RAPPRESENTAZIONE DI GRAFI

Prima di tutto vediamo come possiamo rappresentare un grafo in Python. Nel caso degli alberi abbiamo definito solo una classe per i nodi e la struttura dell'albero era rappresentata dai legami nodo-figli contenuti in ogni nodo. Per i grafi potremmo adottare la stessa tecnica ma risulterebbe poco agevole, perché in un grafo non c'è un nodo particolare come negli alberi da cui è possibile visitare tutti gli altri.

Per queste ragioni, rappresenteremo il grafo utilizzando una classe che gestisce l'intero grafo e una classe senza metodi usata per memorizzare le informazioni dei nodi. I nodi sono identificati da un nome che potrà essere un qualsiasi valore immutabile, ad esempio una stringa, un intero o una tupla. Per rappresentare gli archi possiamo mantenere per ogni nodo un insieme di tutti i nodi che gli sono connessi, detti *adiacenti*. Per mantenere i nodi e trovare velocemente un nodo dato il suo nome, utilizziamo un dizionario.

La classe `Graph` utilizza un dizionario che ad ogni nome di nodo associa un oggetto `_GraphNode` che contiene il nome `name`, l'insieme degli adiacenti `adj`, e la posizione del nodo `pos` utilizzata per poter visualizzare il grafo. In questo modo non ci possono essere due nodi con lo stesso nome. La classe `_GraphName` è una classe di convenienza a cui l'utente non può accedere. Un grafo sarà creato dai metodi `addEdge(name)` per aggiungere un nodo e `addEdge(name1,name2)` per aggiungere un arco. Per interrogare il grafo, utilizzeremo i metodi `nodes()` per la lista dei nodi, `adjacent(name)` per la lista dei nodi adiacenti a `name`, `edges()` per la lista di tutti gli archi, `pos(name)` per la posizione del nodo e `info(name)` per ottenere le informazioni aggiuntive al nodo `name`. Nei grafo, salveremo anche due colori, per i nodi e per gli archi, da usare durante la visualizzazione. I colori saranno accessibili dal metodo `colors()` e assegnabili dal metodo `setColors()`.

```
class _GraphNode:
    '''Rappresenta un nodo del grafo.
```

```
Da usarsi solo all'interno di Graph.'''
```

```
def __init__(self, name, adj, pos):  
    self.name = name  
    self.adj = set(adj)  
    self.pos = pos
```

```
class Graph:
```

```
    '''Rappresenta un grafo.'''
```

```
def __init__(self, colors=["black", "black"]):  
    '''Inizializza un grafo vuoto.'''  
    self._nodes = {}  
    self._colors = list(colors)
```

```
def addNode(self, name, pos):  
    '''Aggiunge un nodo name, se non esiste'''  
    if name in self._nodes: return  
    self._nodes[name] = _GraphNode(name, set(), pos)
```

```
def addEdge(self, name1, name2):  
    '''Aggiunge un arco che collega i nodi  
    name1 e name2'''  
    if name1 not in self._nodes: return  
    if name2 not in self._nodes: return  
    self._nodes[name1].adj.add(name2)  
    self._nodes[name2].adj.add(name1)
```

```
def adjacents(self, name):  
    '''Ritorna una lista dei nomi dei nodi  
    adiacenti al nodo name, se il nodo non  
    esiste, ritorna None'''  
    if name not in self._nodes: return None  
    return list(self._nodes[name].adj)
```

```
def nodes(self):  
    '''Ritorna una lista dei nomi dei nodi'''  
    return list(self._nodes.keys())
```

```
def edges(self):  
    '''Ritorna una lista degli archi'''  
    edges = set()  
    for name, node in self._nodes.items():
```

```

        for adj in node.adj:
            # salta archi ripetuti
            if (adj, name) in edges:
                continue
            edges.add( (name,adj) )
    return list(edges)

def pos(self, name):
    '''Ritorna la posizione del nodo name'''
    if name not in self._nodes: return None
    return self._nodes[name].pos

def colors(self):
    return list(self._colors)

def setColors(self, colors):
    self._colors = list(colors)

```

Abbiamo usato un `set` per gli adiacenti anziché una lista per motivi di efficienza. Infatti il metodo `addEdge()`, che aggiunge un arco tra due nodi, deve evitare di aggiungere l'arco se è già presente. I `set` garantiscono ciò e in modo molto efficiente. Si osservi che l'arco aggiunto è simmetrico essendo aggiunto sia per la coppia di nodi che per quella opposta.

Il metodo `adjacents(name)` ritorna una lista dei nomi degli adiacenti al nodo `name`. Avrebbe potuto ritornare anche un `set` ma sarebbe dovuto essere comunque una copia dell'insieme originale per evitare che chi usa un oggetto `Graph` possa modificare il grafo senza che l'oggetto `Graph` possa aggiornare la propria rappresentazione. Stesso discorso per il metodo `nodes()` che ritorna anch'esso una lista creata ex novo dei nomi dei nodi.

Possiamo ora creare un semplice grafo che simula una piccolissima rete sociale con 4 persone e 3 relazioni di amicizia, come ad esempio quella vista sotto. Per poter visualizzare i nodi, assegneremo anche le loro posizioni.

```

g = Graph()

# aggiunge i nodi
g.addNode('Sara', (125, 75))
g.addNode('Ciro', (0, 75))

```

```
g.addNode( 'Marco', (225,0))
g.addNode( 'Andrea', (225,125))

# aggiunge gli archi
g.addEdge( 'Ciro', 'Sara')
g.addEdge( 'Marco', 'Sara')
g.addEdge( 'Andrea', 'Sara')

# Interroga il grafo
print(g.nodes())
# Out: ['Sara', 'Ciro', 'Marco', 'Andrea']
print(g.adjacents('Sara'))
# Out: ['Marco', 'Ciro', 'Andrea']
```

## 21.3 VISUALIZZAZIONE DI GRAFI

Per visualizzare i grafi scriviamo una funzione che esporta una lista di grafi come SVG, un formato simile all'HTML, ma che permette di definire una collezione di elementi grafici invece che testuali. Usiamo cerchi per i nodi, linee per gli archi, e un nodo per centrare il grafo nell'immagine.

```
def size_graphs(graphs):
    '''Trova la dimensione di una lista di grafi'''
    w, h = 0, 0
    for g in graphs:
        for node in g.nodes():
            p = g.pos(node)
            if w < p[0]: w = p[0]
            if h < p[1]: h = p[1]
    return w, h

def dump_graph(g):
    '''Crea il codice SVG per il grafo g.'''
    # formati
    circle_fmt = '<circle r="3" cx="{}" cy="{}" fill="{}/>\n'
    line_fmt = '<line x1="{}" y1="{}" x2="{}" y2="{}" stroke="{}"
stroke-width="2"/>\n'
    svg = ''
    node_color, edge_color = g.colors()
    if node_color:
        for name in g.nodes():
            pos = g.pos(name)
            svg += circle_fmt.format(pos[0], pos[1],
                g.colors()[0])
    if edge_color:
        for name1, name2 in g.edges():
            pos1 = g.pos(name1)
            pos2 = g.pos(name2)
            svg += line_fmt.format(pos1[0], pos1[1],
                pos2[0], pos2[1], g.colors()[1])
    return svg
```

```

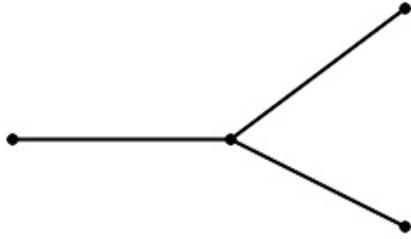
def dump_graphs(graphs):
    '''Crea un'immagine SVG per i grafi graphs.'''
    # trova la dimensione del grafo
    w, h = size_graphs(graphs)
    # formati
    svg_fmt = '<svg xmlns="{}" width="{}" height="{}">\n'
    group_fmt = '<g transform="translate(5,5)">\n'
    # svg namespace
    ns = "http://www.w3.org/2000/svg"
    svg = svg_fmt.format(ns, w+10, h+10)
    svg += group_fmt
    for g in graphs:
        svg += dump_graph(g)
    svg += '</g>\n'
    svg += '</svg>\n'
    return svg

def save_graphs(filename,graphs):
    '''Salva i grafi in SVG sul file filename'''
    with open(filename,'w') as f:
        f.write(dump_graphs(graphs))

print(dump_graphs([g]))
# Out: <svg xmlns="http://www.w3.org/2000/svg" width="235"
height="135">
# Out: <g transform="translate(5,5)">
# Out: <circle r="3" cx="125" cy="75" fill="black"/>
# Out: <circle r="3" cx="0" cy="75" fill="black"/>
# Out: <circle r="3" cx="225" cy="0" fill="black"/>
# Out: <circle r="3" cx="225" cy="125" fill="black"/>
# Out: <line x1="125" y1="75" x2="225" y2="125" stroke="black"
stroke-width="2"/>
# Out: <line x1="125" y1="75" x2="0" y2="75" stroke="black" stroke-
width="2"/>
# Out: <line x1="125" y1="75" x2="225" y2="0" stroke="black"
stroke-width="2"/>
# Out: </g>
# Out: </svg>
# Out:

save_graphs('img_graph00.svg',[g])

```



Implementiamo ora una funzione per visualizzare i grafi in modo interattivo usando il modulo `gwidget`. In questo caso usiamo una variabile globale nascosta per memorizzare i grafi da disegnare ad ogni frame. Aggiungiamo anche una variabile globale per disattivare il disegno interattivo nel caso vogliamo velocizzare l'esecuzione.

```
from PyQt5.QtGui import *
from PyQt5.QtCore import *
from PyQt5.QtWidgets import *
from gwidget import run_app

# grafo usato implicitamente da paint_graph
_paint_graphs = None

# variabile globale per disattivare il
# disegno interattivo
skipui = True

def paint_graph(painter,g):
    '''Disegna un grafo con la libreria Qt.'''
    node_color, edge_color = g.colors()
    if node_color:
        painter.setPen(QColor(node_color))
        painter.setBrush(QColor(node_color))
        for name in g.nodes():
            pos = g.pos(name)
            painter.drawEllipse(
                QPoint(pos[0], pos[1]), 3, 3)
    if edge_color:
        painter.setPen(QColor(edge_color))
        painter.setBrush(QColor(edge_color))
        for name1, name2 in g.edges():
            pos1 = g.pos(name1)
            pos2 = g.pos(name2)
            painter.drawLine(
```

```

        QPoint(pos1[0], pos1[1]),
        QPoint(pos2[0], pos2[1]))

def paint_graphs(painter):
    '''Disegna una lista di grafi con la libreria
    Qt. I grafi sono memorizzati nella variabile
    globale _paint_graphs.'''
    size = painter.info.size
    painter.fillRect(0, 0, size[0], size[1],
        QColor(255,255,255))
    for g in _paint_graphs:
        paint_graph(painter,g)

def view_graphs(graphs):
    '''Visualizza i grafi graphs'''
    # velocizza l'esecuzione saltando la ui
    if skipui: return
    global _paint_graphs
    _paint_graphs = graphs
    w, h = size_graphs(graphs)
    run_app(paint_graphs,w,h)
    _paint_graphs = None

view_graphs([g])

```

## 21.4 LABIRINTI

I grafi possono essere usati per risolvere problemi di navigazione, come calcolare il percorso tra due punti su una mappa o, equivalentemente, risolvere labirinti. Inizieremo come questo secondo problema. Per specificare i labirinti più semplicemente, scriviamo una funziona che prende in input una stringa in cui i corridoi sono spazi e tutti gli altri caratteri sono muri. Un esempio di labirinto memorizzato in questo modo è incluso di seguito.

```
labyrinth = '''
+--+--+--+--+--+--+--+--+--+
|          |          |
+  +--+--+--+--+  +  +  +--+--+  +
|  |  |  |  |  |  |  |  |  |
+--+  +  +--+  +--+--+--+--+  +  +  +
|  |  |  |  |  |  |  |  |  |
+  +--+--+  +--+  +--+  +  +--+  +  +
|          |  |  |  |  |  |  |
+--+--+  +--+  +--+  +  +  +  +--+--+
|          |  |  |  |  |  |  |
+  +--+--+--+--+--+  +  +--+  +--+  +
|  |  |  |  |  |  |  |  |  |
+--+  +  +--+  +  +--+  +--+--+--+  +
|  |  |  |  |  |  |  |  |  |
+--+--+--+--+--+--+--+--+--+
'''
```

Data questa rappresentazione creiamo un grafo i cui nodi sono i corridoi e dove inseriamo archi tra i nodi dei corridoi che sono adiacenti nel testo. I nomi dei nodi sono le tuple  $(x,y)$  delle coordinate dei nodi nel testo. Per poter visualizzare il labirinto meglio, ritorniamo anche il grafo dei muri creato in modo similare.

```
def parse_labyrinth(text):
    '''Creare il grafo dei corridoi e il grafo dei
```

```

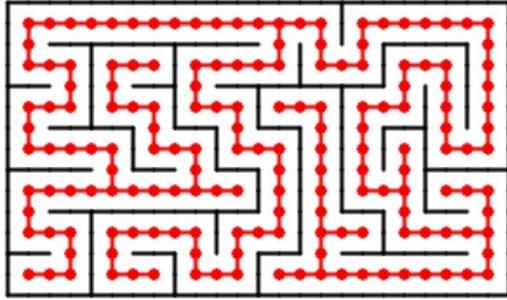
muri a partire da un labirinto testulae.'''
# lista delle linee del testo
lines = text.strip().splitlines()
# Dimensioni del labirinto
w, h = len(lines[0]), len(lines)
# grafo di corridoi e muri
corridors = Graph(['red', 'red'])
walls = Graph(['', 'black'])
# aggiunta dei nodi
for j in range(h):
    for i in range(w):
        # calcola la posizione
        pos = (i*12, j*12)
        # aggiungi il nodo ai corridoi o muri
        if lines[j][i] == ' ':
            corridors.addNode( (i,j), pos )
        else:
            walls.addNode( (i,j), pos )
# aggiunta degli archi
for j in range(h):
    for i in range(w):
        # itera sui possibili vicini
        adj = [(-1,0),(1,0),(0,-1),(0,1)]
        for di, dj in adj:
            # coordinate vicino
            ii, jj = i + di, j + dj
            if ii < 0 or jj < 0: continue
            if ii >= w or jj >= h: continue
            # verifica se entrambi sono corridoi
            if (lines[j][i] == ' ' and
                lines[jj][ii] == ' '):
                corridors.addEdge((i,j),(ii,jj))
            # verifica se entrambi sono muri
            elif (lines[j][i] != ' ' and
                  lines[jj][ii] != ' '):
                walls.addEdge((i,j),(ii,jj))
return corridors, walls

```

```

corridors, walls = parse_labyrinth(labyrinth)
view_graphs([corridors, walls])
save_graphs('img_graph01.svg',[corridors, walls])

```



## 21.5 VISITA DI GRAFI

La *visita* di un grafo è l'esplorazione dei nodi a partire da un nodo scelto e seguendo gli archi per passare da un nodo ad un altro, prendendo le dovute precauzioni per evitare di ritornare su nodi già visitati.

Una delle proprietà più semplici che una visita permette di scoprire è la *connessione* del grafo, cioè, se per ogni due nodi c'è un cammino che parte da uno di essi e arriva all'altro passando per archi e possibilmente altri nodi. Ad esempio, in un grafo di una rete sociale, la connessione del grafo significa che una persona è collegata ad una qualsiasi altra seguendo gli amici, gli amici degli amici, e così via. In un labirinto, è possibile trovare l'uscita solo se il grafo è connesso.

Un'altra proprietà che può essere facilmente determinata con una visita sono le distanze tra i nodi. La *distanza* tra due nodi è il numero minimo di archi che bisogna attraversare per raggiungere uno dei nodi partendo dall'altro. Nelle reti sociali, la distanza è il numero di amici tra due persone. Nei labirinti, la distanza è il numero di mosse da fare per uscire.

## 21.6 VISITA IN AMPIEZZA

La visita che considereremo è la *visita in ampiezza* o *Breadth First Search*. A partire da un nodo, si visitano tutti i nodi adiacenti, poi per ognuno di questi si fa la stessa cosa finché non ci sono più altri nodi da visitare. Bisogna però evitare di visitare nodi già visitati, altrimenti il programma non conclude mai l'esecuzione. Per questo basta salvare in un insieme, che chiameremo `visited`, i nodi già visitati. Così testando se un nodo incontrato durante la visita appartiene a `visited` sapremo se lo abbiamo già visitato. Ci occorre anche sapere quali nodi tra quelli in `visited` sono ancora utili per visitare nuovi nodi, cioè i loro vicini non sono stati ancora esplorati. Questo insieme di nodi lo chiameremo `active` e all'inizio conterrà il nodo di partenza. Ad ogni passo della visita estraiamo un nodo da `active` ed esploriamo i suoi vicini. Ogni volta che troviamo un vicino non ancora visitato, lo salviamo in un altro insieme che chiameremo `newactive`. Quando tutti i nodi in `active` saranno stati esaminati, l'insieme `newactive` prenderà il ruolo di `active` e la visita continua in modo analogo fino a che non ci saranno più nodi in `active`.

Scriviamo ora una funzione che ritorna la lista dei nodi visitati a partire da un certo nodo. La useremo per verificare se il labirinto precedente è risolvibile a partire da  $(1,1)$ , verificando che il labirinto è interamente connesso.

```
def visit(g, name):
    '''Visita (tramite BFS) il grafo g a partire dal
    nodo name e ritorna l'insieme dei nomi dei nodi
    visitati'''
    # Inizializza l'insieme dei visitati
    visited = set([name])
    # Inizializza l'insieme degli attivi
    active = set([name])
    # Finchè ci sono nodi attivi,
    while active:
        # Insieme dei nuovi attivi
        newactive = set()
```

```
# Finchè ci sono nodi attivi,
while active:
    # estrai un nodo da active
    u = active.pop()
    # e per ogni suo vicino,
    for v in g.adjacents(u):
        # se non è già visitato,
        if v not in visited:
            # aggiungilo ai visitati
            visited.add(v)
            # e ai nuovi attivi
            newactive.add(v)
    # I nuovi attivi diventano gli attivi
    active = newactive
return visited
```

```
visited = visit(corridors, (1,1))
print(len(corridors.nodes()), len(visited))
# Out: 167 167
# Il labirinto è connesso a (1,1).
```

## 21.7 VISUALIZZAZIONE DELLA VISITA

Per capire meglio la funzione precedente possiamo visualizzarne l'esecuzione, ad esempio salvando una lista di coppie (active, visited), con una coppia per ogni iterazione del ciclo while active. Dato che le nostre funzioni di visualizzazione prendono come input i grafi, salveremo active e visited come grafi con soli nodi.

```
def make_node_graph(g, names, colors):
    '''Crea un grafo con i nodi in names e le
    posizioni in g'''
    ng = Graph(colors)
    for name in names:
        ng.addNode(name, g.pos(name))
    return ng

def visit_traced(g, name):
    '''Visita (tramite BFS) il grafo g a partire dal
    nodo name e ritorna l'insieme dei nomi dei nodi
    visitati. Traccia l'esecuzione con una lista di
    grafi.'''
    visited = set([name])
    active = set([name])
    # Traccia di esecuzione
    trace = []
    while active:
        # Aggiorna la traccia
        trace += [ (
            make_node_graph(g, visited, ['blue', '']),
            make_node_graph(g, active, ['yellow', '']),
        ) ]
        newactive = set()
        while active:
            u = active.pop()
            for v in g.adjacents(u):
                if v not in visited:
                    visited.add(v)
                    newactive.add(v)
```

```

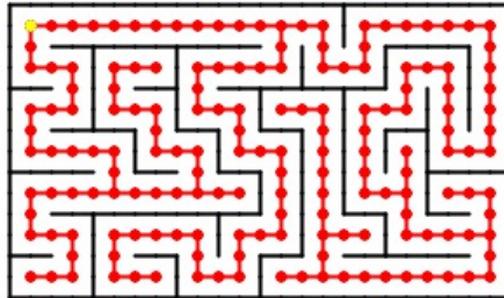
        active = newactive
    return visited, trace

visited, trace = visit_traced(corridors, (1,1))

# visualizziamo le iterazioni
for i in range(len(trace)):
    t = trace[i]
    save_graphs('img_graph01/v{:02}.svg'.format(i),
                [corridors, walls] + list(t))

```

L'immagine di seguito è un'animazione della visita ottenuta dai fotogrammi in precedenza. Su alcuni lettori è necessario premere l'immagine per vedere l'animazione.



Possiamo visualizzare in modo interattivo la visita, modificando la funzioni di visualizzazione per includere una lista di grafi animati.

```

_animated_graphs = None
_frame = 0
_frame_toswitch = 15

def paint_animated_graphs(painter):
    '''Disegna una lista di grafi con la libreria
    Qt. I grafi sono memorizzati nella variabile
    globale _paint_graphs. I grafi da animare sono
    contenuti in _animated_graphs.'''
    size = painter.info.size
    painter.fillRect(0, 0, size[0], size[1],
                    QColor(255,255,255))
    for g in _paint_graphs:
        paint_graph(painter,g)

```

```

global _frame
_frame += 1
animateid = ((_frame // _frame_toswitch) %
             len(_animated_graphs))
for g in _animated_graphs[animateid]:
    paint_graph(painter,g)

def view_animated_graphs(graphs, animated):
    '''Visualizza i grafi graphs e i grafi animati
    animated_graphs'''
    if skipui: return
    global _paint_graphs, _animated_graphs
    _paint_graphs = graphs
    _animated_graphs = animated
    w, h = size_graphs(graphs)
    run_app(paint_animated_graphs,w,h)
    _paint_graphs = None
    _animated_graph = None

view_animated_graphs([corridors, walls],trace)

```

Facciamo un'altro esempio togliendo molti muri dal labirinto.

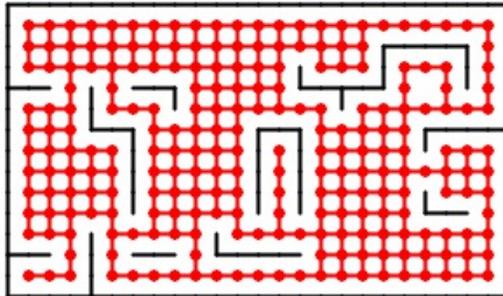
```

labyrinth1 = '''
+--+--+--+--+--+--+--+--+--+--+
|                                     |
+                                     +--+--+ +
|                                     | | | |
+--+ +--+ +--+ +--+--+ + + +
| | | | | | | | | |
+ +--+ +--+ +--+ +--+--+
| | | | | | | | | |
+ | | | | | | | | +
| | | | | | | | | |
+--+ +--+ +--+ +--+--+ +
| | | | | | | | | |
+--+--+--+--+--+--+--+--+--+
'''

corridors1, walls1 = parse_labyrinth(labyrinth1)

```

```
save_graphs('img_graph02.svg',[corridors1, walls1])
view_graphs([corridors1, walls1])
```

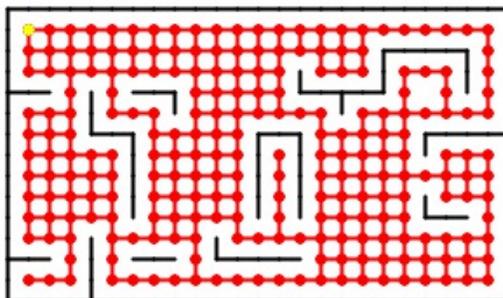


Visualizziamo ora la vista su questo grafo. Notiamo come l'algoritmo si espande "a macchia d'olio".

```
visited, trace = visit_traced(corridors1, (1,1))

# visualizziamo alcuni step
for i in range(len(trace)):
    t = trace[i]
    save_graphs('img_graph02/v{:02}.svg'.format(i),
                [corridors1, walls1] + list(t))
    view_graphs([corridors1, walls1] + list(t))

# visualizziamo l'animazione
view_animated_graphs([corridors1, walls1],trace)
```



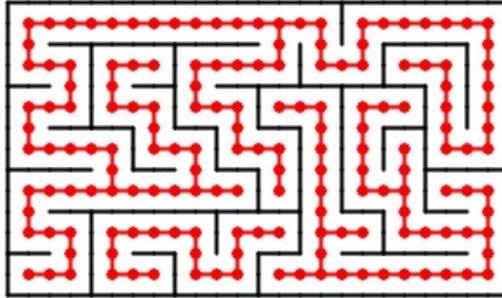
## 21.8 SOTTOGRAFI

Modifichiamo adesso il labirinto introducendo alcuni muri e ripetiamo le operazioni precedenti, notando ora che questo secondo labirinto non è connesso.

```
labyrinth2 = '''
+--+--+--+--+--+--+--+--+--+
|          |          |
+  +--+--+--+--+  +  +  +--+--+  +
|  |  |  |  |  |  |  |  |  |
+--+  +  +--+  +--+--+--+--+  +  +
|  |  |  |  |  |  |  |  |
+  +--+--+  +--+  +--+  +  +--+  +  +
|          |  |  |  |  |  |
+--+--+--+--+--+--+  +--+  +--+  +
|  |  |  |  |  |  |
+--+  +  +--+  +  +--+  +--+--+--+  +
|  |  |  |  |
+--+--+--+--+--+--+--+--+--+--+
'''

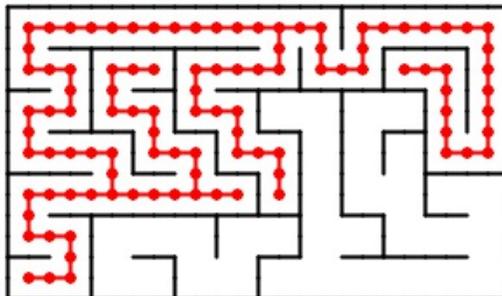
corridors2, walls2 = parse_labyrinth(labyrinth2)
view_graphs([corridors2, walls2])
save_graphs('img_graph03.svg',[corridors2, walls2])

visited2 = visit(corridors2, (1,1))
print(len(corridors2.nodes()),len(visited2))
# Out: 165 103
```



Per visualizzare più chiaramente quali sono le zone raggiungibili da un certo punto, estraiamo un sottografo a partire da un grafo e un insieme di nodi. Il sottografo sarà il grafo che ha come nodi i nodi specificati e come archi tutti gli archi del grafo originale che collegano i nodi specificati.

```
def subgraph(g, names):  
    '''Ritorna il sottografo di g relativo ai nodi  
    in names'''  
    subg = Graph(g.colors())  
    for name in names:  
        pos = g.pos(name)  
        subg.addNode(name, pos)  
    for name in names:  
        for a in g.adjacents(name):  
            if a in names:  
                subg.addEdge(name, a)  
    return subg  
  
sub_corridors2 = subgraph(corridors2,visited2)  
view_graphs([sub_corridors2, walls2])  
save_graphs('img_graph04.svg',  
            [sub_corridors2, walls2])
```



## 21.9 COMPONENTI CONNESSE

Se un grafo non è connesso sarà formato da due o più sottografi che sono connessi, dette *componenti connesse*. Abbiamo visto in precedenza un esempio di componente connessa, ottenuta estraendo un sottografo da un grafo iniziale. Scriveremo ora una funzione che estrae tutte le componenti connesse da un grafo. Per farlo iteriamo le funzione `visit()` e `subgraph()` a partire da un grafo iniziale e sottraendone iterativamente le componenti trovate.

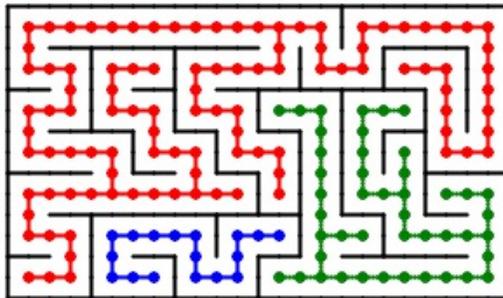
```
def subcomponents(g):
    '''Ritorna le component connesse di g'''

    # lista delle componenti connesse
    components = []
    # sottografo rimasto, all'inizio g
    todo = g
    # finchè il grafo non è vuoto
    while todo.nodes():
        # sceglie un nodo a caso
        startnode = todo.nodes()[0]
        # visita il grafo
        visited = visit(todo,startnode)
        # estrae la component connessa
        component = subgraph(todo,visited)
        # e la aggiunge alla lista
        components += [component]
        # nodi rimanenti
        reminders = set(todo.nodes()) - set(visited)
        # aggiorna il sottografo
        todo = subgraph(todo,reminders)

    # setta i colori da una lista di colori
    colors = ['red', 'green', 'blue', 'orange',
              'yellow', 'magenta', 'purple', 'cyan']
    for i, component in enumerate(components):
        color = colors[i % len(colors)]
```

```
# setta i colori  
component._colors = [color,color]  
return components
```

```
components2 = subcomponents(corridors2)  
view_graphs(components2 + [walls2])  
save_graphs('img_graph05.svg',  
            components2 + [walls2])
```



## 21.10 DISTANZE

Una proprietà interessante della visita in ampiezza è che i nodi sono visitati in ordine di distanza crescente dal nodo di partenza. Infatti, il nodo di partenza è a distanza 0 da sé stesso, poi sono visitati i suoi adiacenti che sono a distanza 1, poi sono visitati gli adiacenti di quest'ultimi che sono a distanza 2, ecc. Possiamo facilmente modificare la funzione `visit()` per calcolare le distanze dei nodi visitati, che manterremo in un dizionario le cui chiavi sono i nomi dei nodi e i valori associati sono le distanze dal nodo di partenza.

```
def distance(g, name):
    '''Ritorna un dizionario che ad ogni nodo
    visitato a partire dal nodo name associa la
    distanza da tale nodo'''
    visited = set([name])
    active = set([name])
    # Dizionario delle distanze
    dist = {name:0}
    while active:
        newactive = set()
        while active:
            u = active.pop()
            for v in g.adjacents(u):
                if v not in visited:
                    visited.add(v)
                    newactive.add(v)
                    # Distanza del nodo visitato
                    dist[v] = dist[u] + 1
            active = newactive
    return dist

distances = distance(corridors,(1,1))
print(distances[(23,13)])
# Out: 58

distances1 = distance(corridors1,(1,1))
```

```
print(distances1[(23,13)])
```

```
# Out: 34
```

```
distances2 = distance(corridors2,(1,1))
```

```
print(distances2[(1,13)])
```

```
# Out: 28
```

```
# non raggiungibile
```

```
distances2 = distance(corridors2,(1,1))
```

```
print((23,13) in distances2)
```

```
# Out: False
```

## 21.11 ALBERO DI VISITA

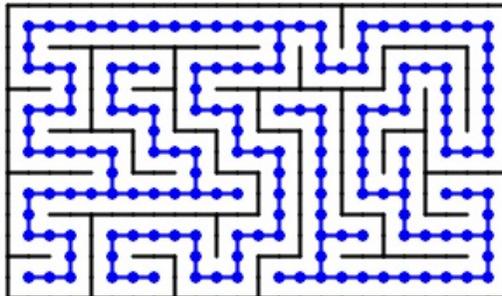
Se vogliamo sapere più dettagli su un cammino, dobbiamo tener traccia di come siamo arrivati a visitare ogni nodo. Così facendo costruiremo implicitamente un albero con radice il nodo di partenza che connette tutti i nodi visitati. Questo è chiamato *albero di visita*. Per costruire l'albero di visita modifichiamo `visit()` registrando per ogni nodo  $u$ , non appena è visitato, il nodo  $v$  che ha permesso di visitarlo. Il nodo  $v$  è il nodo genitore del nodo  $u$  nell'albero di visita. Conviene usare un dizionario che ad ogni nodo visitato associa il suo nodo genitore. La radice, cioè il nodo di partenza, non ha il nodo genitore e quindi gli assoceremo `None`.

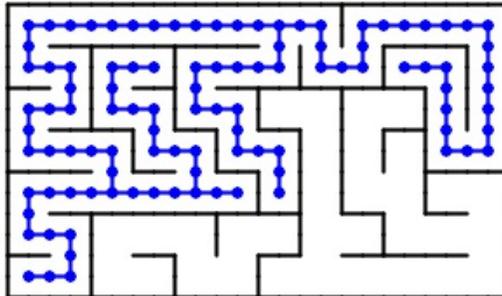
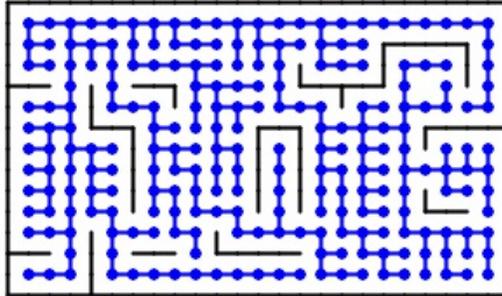
```
def visit_tree(g, name):
    '''Ritorna l'albero di visita tramite un
    dizionario che ad ogni nodo visitato, a partire
    dal nodo name, associa il nome del nodo che lo
    ha scoperto, cioè il nodo genitore.'''
    visited = set([name])
    active = set([name])
    # Albero di visita
    tree = {name:None}
    while active:
        newactive = set()
        while active:
            u = active.pop()
            for v in g.adjacents(u):
                if v not in visited:
                    visited.add(v)
                    newactive.add(v)
                    # Associa al nodo v al genitore
                    tree[v] = u
            active = newactive
    return tree
```

Per visualizzare l'albero della visita, lo possiamo convertire in un grafo i cui i nodi sono i nodi visitati e gli archi sono le relazioni padre-figlio. Con questo

possiamo visualizzare gli alberi di visita dei tre labirinti precedenti.

```
def tree_to_graph(g,tree,colors=['blue','blue']):  
    '''Converte un albero di visita in grafo.'''  
    sg = Graph(colors)  
    for node in tree:  
        sg.addNode(node,g.pos(node))  
    for node, parent in tree.items():  
        if parent:  
            sg.addEdge(node,parent)  
    return sg  
  
tree = visit_tree(corridors,(1,1))  
save_graphs('img_graph06_0.svg',  
            [walls,tree_to_graph(corridors,tree)])  
view_graphs([walls,  
            tree_to_graph(corridors,tree)])  
  
tree1 = visit_tree(corridors1,(1,1))  
save_graphs('img_graph06_1.svg',  
            [walls1,tree_to_graph(corridors1,tree1)])  
view_graphs([walls1,  
            tree_to_graph(corridors1,tree1)])  
  
tree2 = visit_tree(corridors2,(1,1))  
save_graphs('img_graph06_2.svg',  
            [walls2,tree_to_graph(corridors2,tree2)])  
view_graphs([walls2,  
            tree_to_graph(corridors2,tree2)])
```





Dall'albero di visita, rappresentato tramite il dizionario dei genitori, possiamo facilmente ottenere il cammino dal nodo radice a un qualsiasi altro nodo visitato. Basta percorrere a ritroso la sequenza dei nodi genitore dal nodo visitato fino alla radice.

```
def visit_path(tree, name):  
    '''Ritorna una lista contenente il cammino dalla  
    radice al nodo name dell'albero tree  
    rappresentato come dizionario dei genitori'''  
    root = None  
    # Cerca la radice dell'albero  
    for u, gen in tree.items():  
        if gen == None:  
            root = u  
            break  
    # Se è presente nell'albero  
    if name in tree:  
        # Costruisce il cammino risalendo  
        path = [name]  
        # l'albero dal nodo name fino  
        while name != root:  
            # alla radice  
            name = tree[name]  
            path.insert(0, name)
```





## 21.12 GENERAZIONE DI LABIRINTI

Una variazione della funzione di visita può anche essere usata per creare labirinti in modo casuale. Per farlo, costruiamo un sottografo a partire da un grafo connesso. Il sottografo conterrà tutti i nodi, ma solo un sottinsieme di archi. Sceglieremo gli archi facendo una variazione della visita, dove ad ogni iterazione aggiungiamo archi a caso, ma che mantengono il sottografo connesso.

```
def make_grid_graph(w,h):
    '''Crea un grafo fatto a griglia con tutti i
    nodi connessi'''
    g = Graph()
    for j in range(h):
        for i in range(w):
            g.addNode((i,j),(i*20+10,j*20+10))
    adj = [(-1,0),(1,0),(0,-1),(0,1)]
    for i, j in g.nodes():
        for di, dj in adj:
            ii, jj = i + di, j + dj
            if ii < 0 or jj < 0: continue
            if ii >= w or jj >= h: continue
            g.addEdge( (i,j), (ii,jj) )
    return g

import random

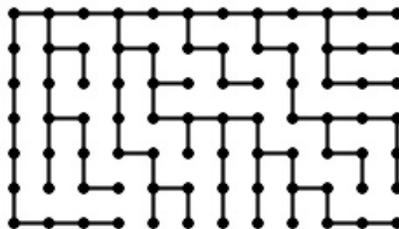
def make_labyrinth(g,name,seed=0):
    '''Crea un labirinto a partire da un grafo
    rimuovendo archi non voluti'''
    # Inizializza un grafo con i nodi di g
    labyrinth = Graph()
    for node in g.nodes():
        labyrinth.addNode(node,g.pos(node))
    # Inizia la visita di g
    visited = set([name])
    # Mantiene una lista di celle adiacenti
    active_adj = g.adjacents(name)
```

```

random.seed(seed)
while active_adj:
    newactive = set()
    # ordine di visita casuale
    random.shuffle(active_adj)
    while active_adj:
        u = active_adj.pop()
        adj = g.adjacents(u)
        # ordine di visita casuale
        random.shuffle(adj)
        # aggiungi u se v è visitato
        for v in adj:
            if v in visited:
                visited.add(u)
                labyrinth.addEdge(u,v)
                break
        # aggiungi tutte gli altri ai candidati
        for v in adj:
            if v not in visited:
                newactive.add(v)
    active_adj = list(newactive)
return labyrinth

labyrinthr = make_labyrinth(make_grid_graph(12,7),
    (0,0), seed=1)
save_graphs('img_graph08_0.svg',[labyrinthr])
view_graphs([labyrinthr])

```



Come vi può vedere, il grafo ha solo gli archi corrispondenti ai corridoi. Se vogliamo fare esempi similari a quelli precedenti, basta convertire il grafo in testo, e poi usare le funzioni già definite. Per farlo, associamo ad ogni nodo ed arco il carattere ' ' ed ad ogni arco mancante i caratteri - quando ci si muove in orizzontale e | quando ci si muove in verticale.

```

def labyrith_to_text(g,w,h):
    '''Converte un grafo di corridoi in testo'''
    txt = '+-' * w + '+\n'
    for j in range(h):
        txt += '| '
        for i in range(1,w):
            if (i-1,j) in g.adjacents((i,j)):
                txt += ' '
            else:
                txt += '| '
        txt += '|\n'
        if j >= h-1: continue
        txt += '+'
        for i in range(w):
            if (i,j+1) in g.adjacents((i,j)):
                txt += ' +'
            else:
                txt += '-+'
        txt += '\n'
    txt += '+-' * w + '+\n'
    return txt

```

```

labyrinthr_text = labyrith_to_text(labyrinthr,12,7)
print(labyrinthr_text)

```

```

# Out: +-+--+--+--+--+--+--+--+--+--+
# Out: |                                     |
# Out: + + +-+ +-+ +-+ +-+ +-+--+
# Out: | |   |   |   |   |   |   |
# Out: + + + + + +-+ +-+ + +-+--+
# Out: | | | | |   |   |   |   |
# Out: + + +-+ + +-+--+--+ +-+--+--+
# Out: | |   | |   |   |   |   |
# Out: + + + + +-+ + + +-+ +-+ +
# Out: | | | |   | | |   |   | |
# Out: + + + +-+ +-+ + + +-+ + +
# Out: | | |   |   | | |   | | |
# Out: + +-+--+--+ + + + + + +-+--+
# Out: |           | | | | | |   |
# Out: +-+--+--+--+--+--+--+--+--+--+
# Out:

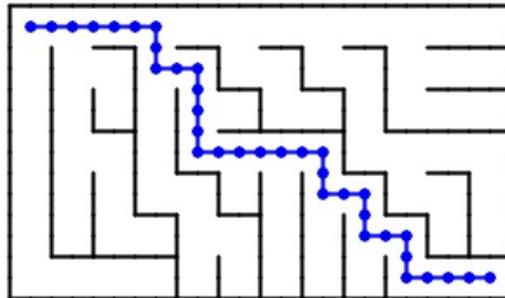
```

```

corridorsr, wallsr = parse_labyrinth(labyrinthr_text)

```

```
treer = visit_tree(corridorsr,(1,1))
pathr = visit_path(treer,(23,13))
save_graphs('img_graph08_1.svg',
            [wallsr, path_to_graph(corridorsr, pathr)])
view_graphs([wallsr,
            tree_to_graph(corridorsr, treer)])
```



## 21.13 MAPPE COME GRAFI DI PIXELS

In questa sezione considereremo grafi di pixel, ovvero grafi i cui nodi sono i pixel di un'immagine e due pixel sono collegati da un arco se sono adiacenti, cioè consecutivi sulla stessa riga o sulla stessa colonna, e hanno colori simili. Questo semplice concetto ci permette di applicare la visita in ampiezza per trovare la strada tra due punti in un'immagine che rappresenta una mappa o un labirinto. Per farlo, scriviamo una funzione che converte l'immagine in un grafo, in modo simile alla funzione che crea un grafo dal testo di un labirinto, e una funzione che altera l'immagine disegnandogli sopra il percorso. Faremo i nostri test su immagini *public domain* da Wikipedia.

```
def similar_pixels(img, i, j, ii, jj, threshold):
    '''Verifica se la differenza in colore tra due
    pixel è inferiore a threshold'''
    w, h = len(img[0]), len(img)
    # verifica se i nodi sono nell'immagine
    if i < 0 or j < 0 or i >= w or j >= h:
        return False
    if ii < 0 or jj < 0 or ii >= w or jj >= h:
        return False
    # calcola la differenza dei colori
    c1 = img[j][i]
    c2 = img[jj][ii]
    diff = (abs(c1[0]-c2[0]) +
            abs(c1[1]-c2[1]) +
            abs(c1[2]-c2[2])) // 3
    return diff <= threshold

def image_to_graph(img, threshold):
    '''Converte un'immagine in un grafo dove i nodi
    sono i pixel dell'immagine e due nodi sono
    adiacenti nel grafo se sono adiacenti
    nell'immagine e la differenza dei colori è
    inferiore a threshold.'''
    g = Graph()
    w, h = len(img[0]), len(img)
```

```

# aggiunge i nodi
for j in range(h):
    for i in range(w):
        g.addNode((i,j),(i,j))
# aggiunge gli archi
for j in range(h):
    for i in range(w):
        # itera sui vicini
        adj = [(-1,0),(1,0),(0,-1),(0,1)]
        for di, dj in adj:
            ii, jj = i + di, j + dj
            # se i colori sono simili,
            if similar_pixels(img, i, j,
                               ii, jj, threshold):
                # aggiungi un arco
                g.addEdge( (i,j), (ii,jj) )

return g

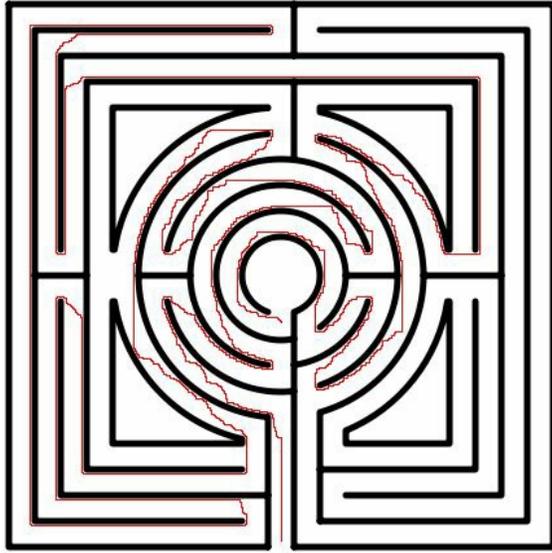
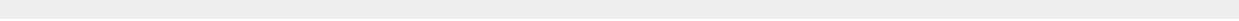
def draw_path(img,path,color=(255,0,0)):
    '''Scrive i pixel in path nell'immagine img con
    colore color.'''
    for i, j in path:
        img[j][i] = color

# funzioni dal capitolo sulle immagini
from image import load, save

def compute_path(infilename, outfilename, start,
                 end, threshold=10):
    '''Altera l'immagine nel file infilename
    colorando il percorso da start a end e
    salvandola nel file outfilename. Usa le
    funzioni load() e save() dal capitolo sulle
    immagini.'''
    img = load(infilename)
    g = image_to_graph(img,threshold)
    tree = visit_tree(g,start)
    path = visit_path(tree,end)
    draw_path(img,path)
    save(outfilename, img)

compute_path('in_maze00.png', 'img_maze00_0.png',
            (215,420), (215,251))

```



## 21.14 VISITA DI GRAFI DI PIXEL

La funzione precedente è piuttosto lenta dato che un'immagine di medie dimensioni può contenere centinaia di migliaia di pixel e la costruzione del grafo è piuttosto onerosa. Non è necessario costruire esplicitamente il grafo perché esso è rappresentato implicitamente dall'immagine stessa. Il fatto che un pixel sia un nodo o meno del grafo dipende solamente dal suo colore che possiamo conoscere dalla posizione  $(x, y)$  del pixel. Inoltre i nodi adiacenti sono ricavabili dalla posizione del pixel considerando il colore dei pixels  $(x-1, y)$ ,  $(x+1, y)$ ,  $(x, y-1)$  e  $(x, y+1)$ . Dobbiamo quindi solamente adattare l'algoritmo di visita che conosciamo a questo tipo di grafo. Dato che usiamo questa versione solo per il calcolo del percorso, possiamo aggiungere anche un argomento opzionale `end` che esce dal ciclo se il punto finale è raggiunto.

```
def visit_tree_image(img, start, end, threshold=10):
    '''Adatta la funzione visit_tree() al grafo
    implicito di un'immagine.'''
    visited = set([start])
    active = set([start])
    tree = {start:None}
    while active:
        newactive = set()
        while active:
            # esce se ha già visitato end
            if end in visited: return tree
            i, j = active.pop()
            # itera sui vicini
            adj = [(-1,0),(1,0),(0,-1),(0,1)]
            for di, dj in adj:
                ii, jj = i + di, j + dj
                # se non sono connessi, continua
                if not similar_pixels(img, i, j,
                    ii, jj, threshold): continue
                if (ii,jj) not in visited:
                    visited.add( (ii,jj) )
                    newactive.add( (ii,jj) )
                    tree[(ii,jj)] = (i,j)
```

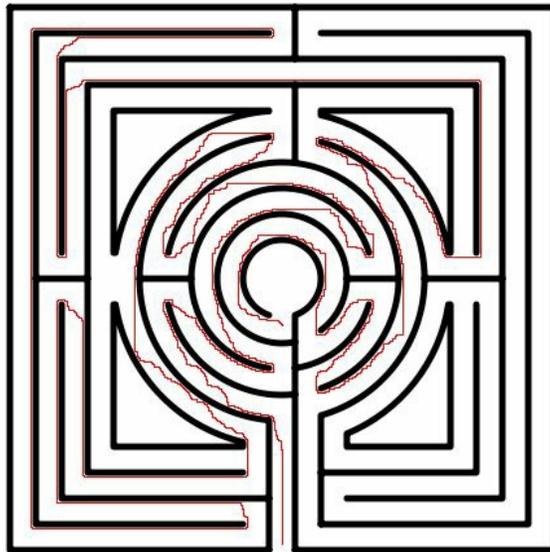
```

        active = newactive
    return tree

def compute_path_image(infile, outfile,
                       start, end, threshold=10):
    '''Altera l'immagine nel file infile
    colorando il percorso da start a end e
    salvandola nel file outfile.
    Usa un grafo implicito.'''
    img = load(infile)
    tree = visit_tree_image(img, start, end, threshold)
    path = visit_path(tree, end)
    draw_path(img, path)
    save(outfile, img)

compute_path_image('in_maze00.png',
                  'img_maze00_1.png', (215,420), (215,251))

```

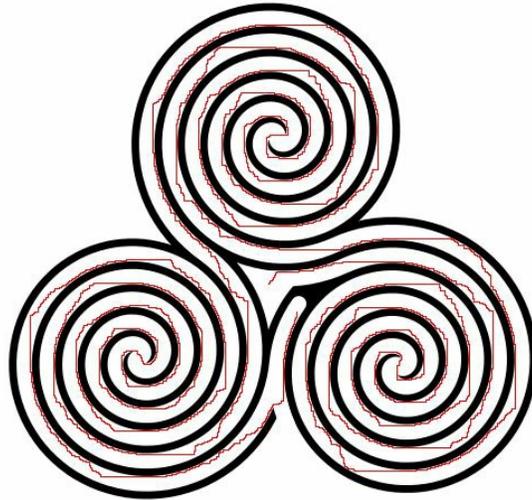


Questa versione è più veloce della precedente, e calcola un percorso che ha la stessa lunghezza del precedente. Le piccole differenze sono dovute all'ordine con cui i vicini vengono considerati che non è lo stesso nei due algoritmi. Applichiamo ora questa funzione ad altre immagini.

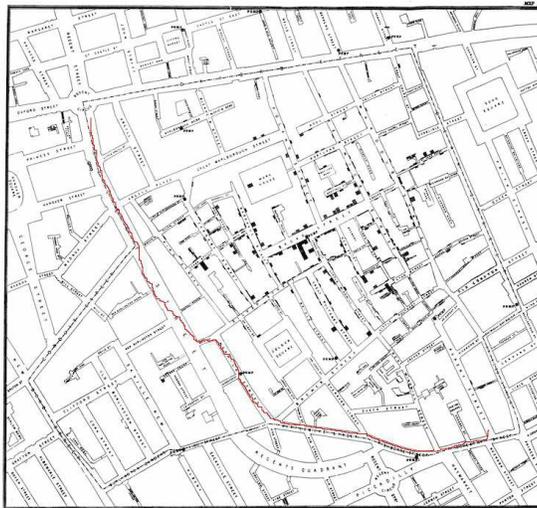
```

compute_path_image('in_maze01.png',
                  'img_maze01_1.png', (270,290), (245,265))

```



```
compute_path_image('in_maze02.png',  
                  'img_maze02_1.png', (130,160), (674,588),25)
```



## 22 WEB CRAWLING

I motori di ricerca come Google hanno una copia locale di tutte le pagine web esistenti in modo da poterle indicizzare. Per fare questa copia ci serve un elenco di tutte le possibili pagine Web in modo da poterle scaricare. Questo non è possibile direttamente perché il Web è decentralizzato nel senso che non esiste un elenco completo delle pagine dato che qualunque persona può creare un pagina web senza dover informare nessuno. Per fare la copia possiamo però trovare le pagine a partire da link in altre pagine. Questo processo è chiamato *Web crawling* e, come vedremo, è equivalente alla visita di un grafo.

## 22.1 GESTIONE DEGLI ERRORI

Nei programmi scritti fino ad ora, gli errori di esecuzione causano la terminazione dell'interprete con la stampa di una stringa di errore. Ci sono però casi in cui gli errori sono legittimi. Ad esempio, potremmo collegarci ad un server Web che è al momento non raggiungibile. In questo caso non sarebbe corretto terminare il programma, ma sarebbe auspicabile *gestire l'errore* quando avviene. Python utilizza il costrutto delle eccezioni per gestire gli errori. Consideriamo un caso semplice dell'accesso ad un elemento non esistente di una lista. Se vogliamo gestire questo errore lo possiamo includere in un blocco `try/except` con la sintassi generale:

```
try:
    istruzioni
except nome_errore1:
    istruzioni_errore1
except nome_errore2:
    istruzioni_errore2
except:
    istruzioni_altri_errori
```

In questo caso Python esegue le istruzioni `istruzioni` e se avviene un errore considera le i blocchi `except`. I blocchi `except nome_errore` vengono eseguiti se avviene l'errore `nome_errore`, mentre le istruzioni nel blocco `except` generico vengono eseguite nel caso di qualunque altro errore.

```
try:
    l = [1,2,3]
    l[10]          # causa un errore
    print('ciao') # non viene eseguita
except:
    print('errore gestito')
# Out: errore gestito
```

## 22.2 WEB CRAWLING

Il *Web crawling* è la visita delle pagine del Web che inizia da una pagina e segue i link ricorsivamente. Il Web crawling si può interpretare come la visita in ampiezza del *grafo del Web*, i cui nodi sono le pagine e gli archi sono i *link* tra le pagine. I link, in questo caso, sono archi asimmetrici dato che una pagina punta verso un'altra, ma non necessariamente il viceversa.

Implementeremo il Web crawling con un modifica della visita sui grafi. Per ogni URL, scarichiamo la pagina e ne estraiamo i link. Questi vengono messi in un insieme di pagine da scaricare successivamente. Per evitare di scaricare più volte la stessa pagina, manteniamo l'insieme degli URL che sono già stati archiviati. In un Web crawler, ad ogni visita si salvano tutte le informazioni scaricate su disco. Nel nostro caso, non salveremo esplicitamente le pagine dato che vogliamo solo simulare il Web crawling e non implementarlo in modo completo.

```
def load_page(url):
    '''Ritorna la pagina HTML dato un URL.'''
    return ''

def get_links(url, html):
    '''Ritorna i links in un HTML.'''
    return []

def web_crawl(url, maxvisits=5):
    '''Visita il grafo del web a partire dall'url
    url e stampa i siti visitati. Visita al massimo
    maxvisits.'''
    # URL visitati
    visited = set()
    # URL da visitare
    active = set([url])
    # Finchè ci sono nodi attivi
    while active and len(visited) < maxvisits:
        # Insieme dei nuovi attivi
```

```
newactive = set()
# Finchè ci sono nodi attivi,
while active and len(visited) < maxvisits:
    # estrai un nodo da active
    url = active.pop()
    # se già visitato, continua
    if url in visited: continue
    # scarica la pagina, aggiungi a visitati
    print('loading page:',url)
    html = load_page(url)
    visited.add(url)
    # verifica se lo scaricamento è ok
    if not html: continue
    # trova i link
    links = get_links(url,html)
    # per ogni pagina linkata e non visitata
    for link in links:
        if link not in visited:
            # aggiungi ai nuovi attivi
            newactive.add(link)
    # I nuovi attivi diventano gli attivi
    active = newactive
return visited
```

## 22.3 SCARICARE UNA PAGINA

Per scaricare una pagina abbiamo visto i metodi predefiniti di Python. Questi funzionano bene per casi semplici, ma nei casi più complessi danno problemi di vario tipo. Per questo utilizzeremo una libreria più ad alto livello chiamata **Requests**, come suggerito dal sito di Python. Per evitare che il programma si interrompa in attesa di un server molto lento, includiamo un timeout di 1 secondo. Per evitare che errori di connessione terminino il programma, includiamo l'intera funzione in un blocco `try/except`.

```
import requests

def load_page(url):
    '''Ritorna la pagina HTML dato un URL.'''
    try:
        page = requests.get(url, timeout=1)
        if page.encoding:
            return page.content.decode(
                page.encoding)
        else:
            return page.content.decode('utf8')
    except:
        return ''

sites = web_crawl('http://python.org')
# Out: loading page: http://python.org
```

## 22.4 ELENCO DEI LINKS

Per trovare i link dobbiamo convertire la pagina in un albero HTML e trovare i tag dei link. Potremmo usare le classi create nei capitoli precedenti, ma queste causerebbero molti errori dato che la maggioranza di documenti HTML nel Web non è ben formata. Per questo usiamo la libreria `lxml`. L'albero del documento sarà di tipo `xml.etree.ElementTree`, che ha il metodo `iter(tag)` per iterare su tag specifici, nel nostro i tag `a`. Useremo `urljoin()` in `urllib.parse` per risolvere i link da locali a globali. Ancora una volta includiamo la funzione in un blocco `try/except` per evitare che errori nella pagine possano interrompere il programma.

```
import html5lib
import urllib

def get_links(url,html):
    '''Ritorna i links in un HTML.'''
    try:
        links = []
        document = html5lib.parse(html,
            namespaceHTMLElements=False)
        for link_elem in document.iter('a'):
            link = link_elem.get('href')
            # Manca l'url del link
            if not link: continue
            # ignoriamo referenze interne
            if '#' in link: continue
            # link locale, aggiungiamo l'url
            if '://' not in link:
                link = urllib.parse.urljoin(url,
                    link)
            # ci sono ancora errori, saltiamo
            if '://' not in link: continue
            # normalizziamo url
            link = urllib.parse.urljoin(link, '.')
            links += [link]
    return links
```

```
except:  
    return []
```

```
sites = web_crawl('http://python.org',maxvisits=15)  
# Out: loading page: http://python.org  
# Out: loading page:  
http://www.riverbankcomputing.co.uk/software/pyqt/  
# Out: loading page: http://jobs.python.org/  
# Out: loading page: http://python.org/users/membership/  
# Out: loading page: http://pythonmentors.com/  
# Out: loading page: http://python.org/psf/donations/  
# Out: loading page: http://python.org/downloads/mac-osx/  
# Out: loading page: https://pypi.python.org/  
# Out: loading page: http://plus.google.com/  
# Out: loading page: https://www.openstack.org/  
# Out: loading page:  
http://feedproxy.google.com/~r/PythonInsider/~3/1zUlkKxW27U/  
# Out: loading page: http://python.org/community/merchandise/  
# Out: loading page: http://python.org/community/diversity/  
# Out: loading page: http://python.org/privacy/  
# Out: loading page: http://python.org/success-stories/
```