PYTHON
PER RAGAZZI

UN'INTRODUZIONE GIOCOSA ALLA PROGRAMMAZIONE

JASON R. BRIGGS

EDIZIONI
LSWR

INDICE IN BREVE

L'autore, l'illustratore, i revisori tecnici
Ringraziamenti
Introduzione

PARTE I: IMPARARE A PROGRAMMARE

1: Non tutti i serpenti strisciano
2: Calcoli e variabili
3: Stringhe, liste, tuple e mappe
4: Disegnare con le tartarughe
5: Porre domande con If e Else
6: Girare in tondo
7: Riciclare il codice con funzioni e moduli
8: Come si usano classi e oggetti
9: Le funzioni interne di Python
10: Moduli utili di Python
11: Ancora grafica della tartaruga
12: Grafica migliore con tkinter

PARTE II: BOUNCE!

13: Iniziamo il primo gioco: Bounce!
14: Completiamo il primo gioco: Bounce!

PARTE III: L'AVVENTUROSA FUGA DI MR. STICK MAN

15: Creare la grafica per il gioco di Mr. Stick Man
16: Sviluppare il gioco di Mr. Stick Man
17: Creare Mr. Stick Man
18: Completare il gioco di Mr. Stick Man

Postfazione: Da qui dove si va?
Appendice: Le parole chiave di Python
Glossario

INTRODUZIONE

Perché imparare a programmare?

La programmazione favorisce la creatività, il ragionamento e la capacità di risolvere problemi. Chi programma ha la possibilità di creare qualcosa dal nulla, di usare la logica per trasformare i costrutti della programmazione in una forma che un computer può eseguire e, quando le cose non vanno proprio come avrebbero dovuto, di usare la capacità di risolvere problemi per stabilire che cosa sia andato storto.

Programmare è un'attività divertente, a volte impegnativa (e ogni tanto anche un po' frustrante) e le capacità che si apprendono programmando possono essere utili sia a scuola sia nel lavoro… anche se quel che fate non ha nulla a che vedere con i computer.

Infine, se non altro, la programmazione è un modo eccellente per passare un pomeriggio quando fuori fa brutto tempo.

PERCHÉ PYTHON?

Python è un linguaggio di programmazione facile da imparare, ma con alcune caratteristiche davvero molto utili per chi è alle prime armi. Il codice è molto facile da leggere, rispetto a quello di altri linguaggi di programmazione, e ha una interfaccia interattiva (la "shell") in cui si possono scrivere i programmi e vederli girare. Oltre alla semplicità della struttura del linguaggio e alla shell interattiva con cui si può sperimentare, Python ha alcune altre caratteristiche che rafforzano di molto il processo di apprendimento e permette di creare semplici animazioni per creare i propri giochi. Uno di questi elementi è il modulo turtle, ispirato alla grafica della tartaruga (utilizzato dal linguaggio di programmazione Logo già negli anni Sessanta) e pensato per le applicazioni in campo educativo. Un altro è il modulo tkinter, un'interfaccia per Tk, un "toolkit", cioè un insieme di strumenti, per le interfacce grafiche, che mette a disposizione un metodo semplice per creare programmi con grafica e animazioni un po' più avanzate.

COME IMPARARE A SCRIVERE CODICE

Come sempre, quando si prova a fare qualcosa per la prima volta, è meglio iniziare dalle basi, perciò cominciate con i primi capitoli e non fatevi prendere dalla tentazione di saltare fino ai capitoli più avanzati. Nessuno è in grado di suonare una sinfonia la prima volta che prende in mano uno strumento musicale. Gli aspiranti piloti non fanno volare un aereo prima di aver capito i controlli fondamentali e i ginnasti (di solito) non sono capaci di fare un salto mortale al primo tentativo. Se andate avanti troppo rapidamente, non solo non vi resteranno in testa le idee fondamentali, ma i contenuti dei capitoli successivi vi sembreranno più complicati di quel che sono in realtà.

Nel procedere, provate tutti gli esempi: così potrete vedere come funzionano. Alla fine della maggior parte dei capitoli ci sono anche dei rompicapo di programmazione che potete cercare di risolvere: vi aiuteranno a migliorare le vostre abilità di programmazione. Ricordate: quanto meglio assimilerete le basi, tanto più facile vi risulterà capire idee più complicate in seguito.

Quando incontrate qualcosa di frustrante o di troppo impegnativo, ecco alcune cose che io trovo utili:

1. Suddividete un problema in parti più piccole. Cercate di capire che cosa fa un piccolo frammento di codice, oppure ragionate solo su una piccola parte di un'idea difficile (concentratevi su un piccolo frammento di codice invece di cercare di capire il tutto in una volta sola).
2. Se questo ancora non vi è d'aiuto, a volte la cosa migliore è semplicemente non pensarci per un po'. Dormiteci sopra e riprendete il problema il giorno dopo. È un buon modo per risolvere molti problemi e può essere particolarmente utile per la programmazione.

CHI DOVREBBE LEGGERE QUESTO LIBRO

Questo libro è per chiunque sia interessato alla programmazione, bambino o adulto che a questo campo si avvicini per la prima volta. Se volete imparare a scrivere il vostro software anziché usare solamente i programmi scritti da altri, questo libro è un ottimo punto di partenza.

Nei capitoli seguenti, troverete le informazioni per installare Python, eseguire la shell e svolgere qualche calcolo, stampare il testo che appare sullo schermo, creare liste e svolgere alcune semplici operazioni di controllo del flusso utilizzando gli enunciati if e i cicli for (e scoprirete che cosa sono gli enunciati if e i cicli for!). Vedrete come riutilizzare il codice con le funzioni, gli elementi fondamentali di classi e oggetti e la descrizione di alcune funzioni e alcuni moduli, fra i molti che fanno parte di Python.

Troverete capitoli sulla grafica della tartaruga, a livello più semplice e più avanzato, e anche sull'uso del modulo `tkinter` per disegnare sullo schermo. Alla fine di molti capitoli vi sono rompicapo di programmazione di varia complessità, che vi aiuteranno a consolidare le conoscenze appena acquisite, con la possibilità di scrivere voi stessi piccoli programmi.

Una volta acquisita la conoscenza dei fondamenti della programmazione, imparerete come scrivere i vostri giochi. Svilupperemo due giochi grafici e scopriremo il rilevamento delle collisioni, gli eventi e varie tecniche di animazione.

La maggior parte degli esempi nel libro usa la shell IDLE (*Integrated DeveLopment Environment*, ambiente integrato di sviluppo) di Python. IDLE evidenzia la struttura sintattica, offre funzioni di copia e incolla (analoghe a quelle che usereste in altre applicazioni) e una finestra di editor in cui potete salvare il vostro codice per poterlo usare ancora in seguito, il che significa che IDLE funziona sia come un ambiente interattivo per la sperimentazione e anche un po' come un editor di testo. Gli esempi funzioneranno altrettanto bene con la console standard e un normale editor di testo, ma l'evidenziazione della sintassi e un ambiente un po' più amichevole come quello di IDLE possono essere di aiuto per capire meglio, perciò nel primo capitolo vedremo come impostarlo.

CHE COSA C'È IN QUESTO LIBRO

Ecco una breve panoramica di quello che troverete in ciascun capitolo.

Il **Capitolo 1** è un'introduzione alla programmazione, con le istruzioni per installare Python.

Il **Capitolo 2** introduce i calcoli fondamentali e le variabili, e il **Capitolo 3** descrive alcuni dei tipi fondamentali di Python, come le stringhe, le liste e le ennuple.

Con il **Capitolo 4** arriva il primo assaggio del modulo `turtle`. Passeremo dagli elementi base della programmazione a come far muovere una tartaruga (sotto forma di una freccia) in giro per lo schermo.

Il **Capitolo 5** tratta delle variazioni delle condizioni e degli enunciati `if` (condizionali); il **Capitolo 6** procede con i cicli `for` e `while`.

Nel **Capitolo 7** inizieremo a usare e creare funzioni, poi nel **Capitolo 8** parleremo di classi e oggetti. Esamineremo le idee base abbastanza a fondo per poter poi utilizzare qualcuna delle tecniche di programmazione necessarie, nei capitoli più avanti, per lo sviluppo di giochi. A questo punto il materiale comincia a diventare un po' più complicato.

Il **Capitolo 9** tratta la maggior parte delle funzioni interne a Python e il **Capitolo 10** continua affrontando alcuni moduli (fondamentalmente, blocchi di funzionalità utili), installati per impostazione predefinita insieme a Python.

Il **Capitolo 11** ritorna al modulo `turtle` per sperimentare qualche forma più complicata. Il **Capitolo 12** passa all'uso del modulo `tkinter` per creare qualche oggetto grafico più avanzato.

Nei **Capitoli 13** e **14** creeremo il nostro primo gioco, "Bounce!", che si fonda sulle conoscenze acquisite nei capitoli precedenti; nei **Capitoli 15-18** creeremo un altro gioco, "L'avventurosa fuga di Mr. Stick Man". I capitoli dedicati allo sviluppo dei giochi sono quelli in cui le cose possono farsi davvero serie. Se incontrate problemi che non riuscite a risolvere altrimenti, scaricate il codice dal sito web che accompagna il libro (*http://python-for-kids.com/*) e confrontate il vostro codice con questi esempi, che sicuramente funzionano.

Nella **Postfazione**, concludiamo dando uno sguardo a PyGame e ad alcuni altri linguaggi di programmazione molto diffusi.

Infine, nell'**Appendice**, potrete trovare le parole chiave di Python trattate in dettaglio e, nel **Glossario**, troverete le definizioni dei termini della programmazione utilizzati in tutto il libro.

IL SITO WEB DI ACCOMPAGNAMENTO

Se vi trovate nelle condizioni di aver bisogno di aiuto mentre leggete, provate a consultare il sito di accompagnamento, *http://python-for-kids.com/* (in inglese), dove potrete trovare tutti gli esempi del libro e altri rompicapo di programmazione. Troverete anche le soluzioni di tutti i rompicapo nel libro, nel caso vi troviate in difficoltà o vogliate controllare il vostro lavoro.

BUON DIVERTIMENTO!

Ricordate, mentre procedete, che la programmazione può essere divertente. Non prendetela come un lavoro. Pensate la programmazione come un modo per creare giochi o applicazioni divertenti che potete condividere con gli amici o con altri.

Imparare a programmare è un meraviglioso esercizio mentale e i risultati possono essere molto gratificanti. Ma soprattutto, qualsiasi cosa facciate, divertitevi!

PARTE I

IMPARARE A PROGRAMMARE

1
NON TUTTI I SERPENTI STRISCIANO

Un programma per computer è un insieme di istruzioni grazie alle quali un computer svolge qualche attività. Non si tratta delle parti fisiche di una macchina (come i fili, i microchip, le schede, il disco fisso e altre cose simili) ma di quella materia impalpabile e nascosta che “gira” su quell’hardware. Un programma per computer, che chiamerò in genere semplicemente *programma*, è l’insieme dei comandi che dicono a quell’hardware “stupido” che cosa fare. *Software* è un insieme di programmi.

Senza programmi, quasi ogni dispositivo che usate quotidianamente smetterebbe di funzionare o sarebbe molto meno utile di quel che è. I programmi, in una forma o in un'altra, controllano non solo il vostro personal computer ma anche i sistemi per i videogiochi, i telefoni cellulari e il navigatore satellitare nell'automobile. Il software controlla anche apparecchi meno ovvi, come i televisori a LCD e i loro telecomandi, radio, lettori di DVD, forni e anche qualche frigorifero dei più recenti. Anche i motori delle automobili, i semafori, i lampioni, i segnali dei treni, i tabelloni elettronici e gli ascensori sono controllati da programmi.

I programmi sono un po' come pensieri. Se non aveste pensieri, probabilmente ve ne stareste seduti sul pavimento, con lo sguardo perso nel vuoto e la saliva che vi gocciola sulla maglietta. Il pensiero "alzati dal pavimento" è un'istruzione, o un comando, che dice al vostro corpo di alzarsi. Allo stesso modo, i programmi dicono ai computer che cosa fare.

Se sapete scrivere programmi per computer, potete fare ogni genere di cose utili. Certo, magari non sarete in grado di scrivere programmi che controllano auto, semafori o il frigorifero (o almeno, non inizialmente), ma potrete creare pagine web, scrivere i vostri giochi o addirittura un programma che vi sia d'aiuto nei compiti a casa.

QUALCHE PAROLA SUL LINGUAGGIO

Come gli esseri umani, i computer possono usare molte lingue per comunicare – in questo caso, linguaggi di programmazione. Un linguaggio di programmazione è solo un modo particolare di parlare a un computer – un modo per usare istruzioni che sia gli esseri umani sia i computer sono in grado di comprendere.

Esistono linguaggi di programmazione che prendono il nome da persone (come Ada e Pascal), quelli che hanno come nome semplici acronimi (come BASIC e FORTRAN) e anche qualcuno che prende il nome da trasmissioni televisive, come Python. Sì, il linguaggio di programmazione Python ha preso il nome dalla trasmissione televisiva inglese *Monty Python's Flying Circus*, non dal serpente (*python* è la parola inglese che significa "pitone", se non l'avevate ancora capito).

Parecchie caratteristiche fanno di Python un linguaggio di programmazione estremamente utile per chi è alle prime armi. Potete utilizzare Python per scrivere molto rapidamente programmi semplici ma molto efficaci. Non usa molti simboli complicati, a differenza di altri linguaggi di programmazione, ed è perciò più facile da leggere e molto più "amichevole" per chi è agli inizi. (Questo non vuol dire che Python non usi simboli – semplicemente il loro uso non è così esteso come in molti altri linguaggi.)

INSTALLARE PYTHON

L'installazione di Python è molto semplice. Qui vedremo come lo si installa in

Windows 10, Mac OS X e Ubuntu. Installando Python, troverete anche un collegamento al programma IDLE (che significa Integrated DeveLopment Environment, ambiente di sviluppo integrato), che permette di scrivere programmi per Python. Se Python è già installato sul vostro computer, potete passare subito al paragrafo "Dopo aver installato Python" a pagina 10.

INSTALLARE PYTHON IN WINDOWS 10

Per installare Python per Microsoft Windows 10, usate il browser web, andate all'indirizzo *http://www.python.org/* e scaricate il programma di installazione più recente per Python 3. Nel menu del sito troverete una sezione intitolata **Downloads**. Un clic sul nome di questa sezione vi porterà a una finestra come questa:

NOTA

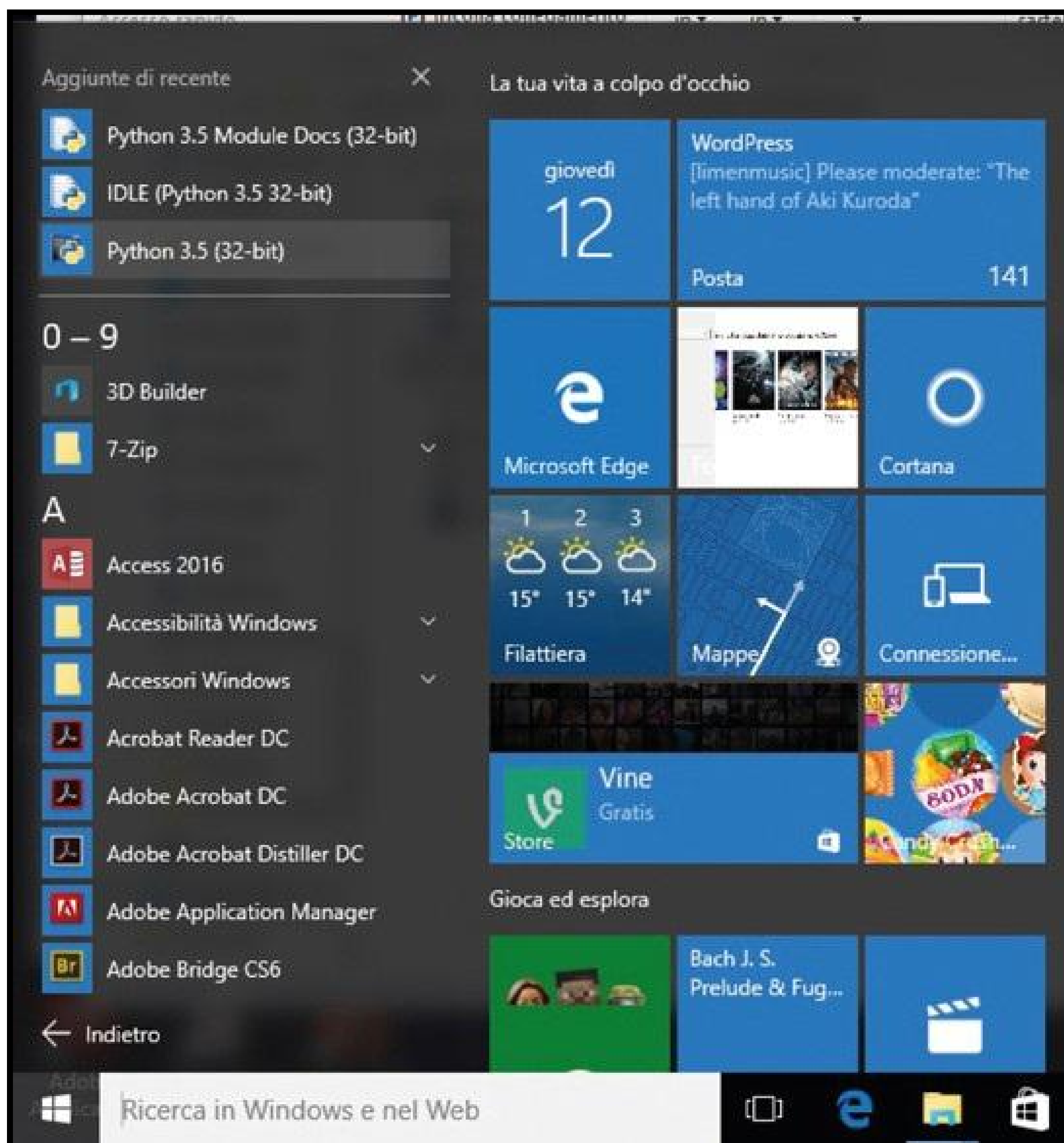
Quale sia esattamente la versione di Python che scaricate non è importante, basta che il suo numero inizi con il 3. Gli aggiornamenti sono frequenti: qui vedete le indicazioni per la versione 3.5.1, ma è probabile che quando leggerete queste pagine la versione più recente abbia un numero più alto.

Una volta scaricato il programma di installazione per Windows, fate un doppio clic sulla sua icona, poi seguite le istruzioni per installare Python nella posizione predefinita.

1. Selezionate la casella di controllo **Install launcher for all users** (se non è già selezionata).
2. Non cambiate la directory di destinazione predefinita, ma annotatevi il nome della directory (potrebbe essere C:\Python32, per esempio).
3. Ignorate la sezione *Customize installation* e fate clic su **Install now**.

Alla fine del procedimento, nel menu Start troverete, tra le nuove app, o fra le

app installate di recente, alcune voci relative a Python.



Poi, seguite questi passi per aggiungere un collegamento a Python 3 al vostro desktop:

1. Fate clic destro in un punto vuoto del desktop, poi selezionate **Nuovo, Collegamento** dal menu di scelta rapida.
2. Nella casella sotto la scritta **Immettere il percorso per il collegamento**, scrivete (assicurandovi che il nome della directory sia quello che vi siete annotati in precedenza):

```
c:\Python\32\Lib\idlelib.idle.pyw -n
```

3. Fate clic su **Avanti** per passare alla finestra di dialogo successiva.
4. Inserite il nome come *IDLE* e fate clic su **Fine** per creare il collegamento.

Ora potete passare a "Dopo avere installato Python" a pagina 10 per iniziare.

INSTALLARE PYTHON SU MAC OS X

Se usate un Mac, dovreste avere una versione di Python già preinstallata, ma probabilmente si tratta di una versione più vecchia. Per essere sicuri di avere la più recente, aprite il browser e andate all'indirizzo *http://www.python.org/getit/* per scaricare il programma di installazione più recente per il Mac.

I programmi di installazione sono diversi: quello che dovete scaricare dipende dalla versione di Mac OS X che avete. (Se avete dubbi, potete scoprire il numero di versione del sistema operativo facendo clic sulla Mela nella barra dei menu in alto e scegliere Informazioni su questo Mac.)

Se il vostro computer ha una versione di Mac OS X compresa fra la 10.3 e la 10.6, scaricate la versione a 32 bit di Python 3 per i386/PPC.

Se il vostro computer ha una versione di Mac OS X 10.6 o superiore, scaricate la versione 64-bit/32-bit di Python 3 per x86-64.

Una volta scaricato il file (avrà un nome con estensione *.pkg* e lo troverete nella cartella Download), fate un doppio clic su di esso e partirà il processo guidato di installazione.

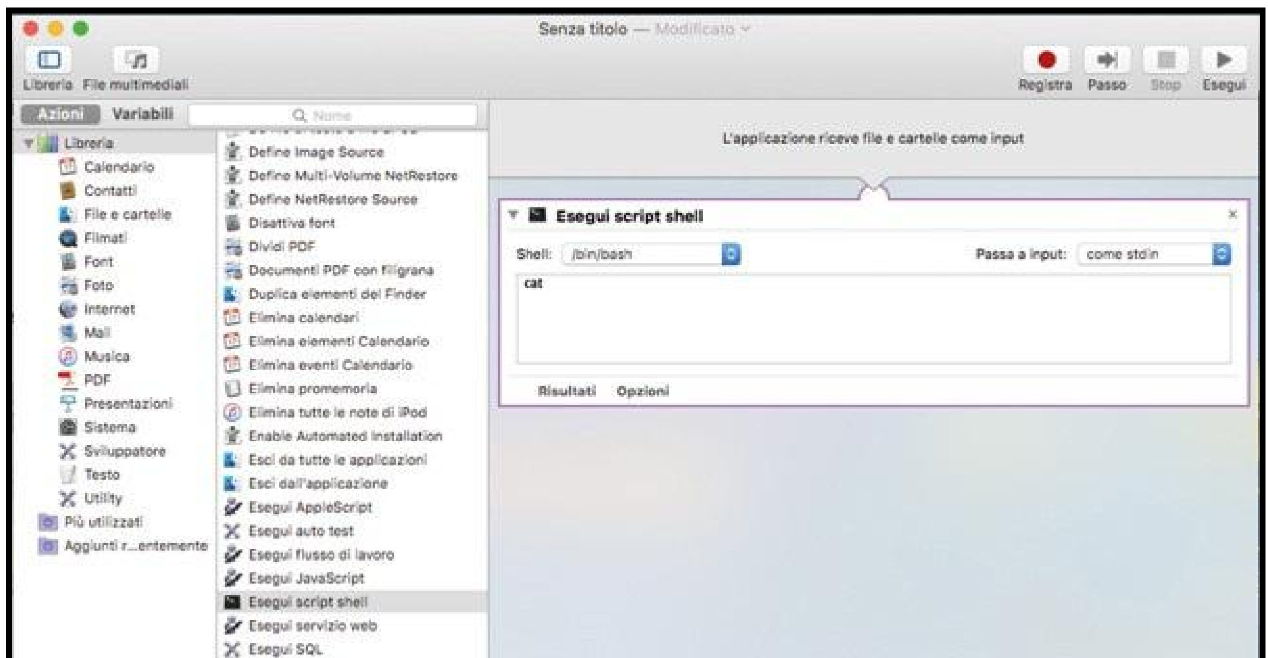
In questa finestra, fate doppio clic su *Python mpkg*, poi seguite le istruzioni per installare il software. Vi verrà chiesta la password di amministratore del vostro Mac. (Non conoscete la password di amministratore? Probabilmente dovrete chiedere ai vostri genitori di inserirla.)

Poi, dovete aggiungere al desktop uno script per lanciare l'applicazione IDLE di Python, in questo modo:

1. Fate clic sull'icona **Spotlight**, la lente di ingrandimento nell'angolo superiore destro della schermata.
2. Nella finestra che compare, scrivete *Automator*.

3. Fate clic sull’applicazione che ha l’aspetto di un robot, quando compare nel menu. Sarà nella sezione intitolata Più utilizzati o in **Applicazioni**.
4. Avviato Automator, selezionate il modello Applicazione:

5. Fate clic su **Scegli** per continuare.
6. Nell’elenco delle azioni, trovate **Esegui script shell** e trascinatelo nel riquadro vuoto a destra. Vedrete qualcosa di simile a questo:



7. Nella casella di testo, vedrete la parola *cat*. Selezionate questa parola e

sostituitela con il testo seguente (tutto, da `open` a `-n`):

```
open -a "/Applications/Python 3.2/IDLE.app" -args -n
```

Può darsi che dobbiate cambiare la directory, a seconda della versione di Python che avete installato.

8. Selezionate **File, Salva** e inserite come nome *IDLE*.
9. Selezionate **Desktop** dalla finestra di dialogo Situato in, poi fate clic su **Salva**.

Ora potete passare a “Dopo avere installato Python” a pagina 10 per iniziare.

INSTALLARE PYTHON SU UBUNTU

Python è già preinstallato nella distribuzione Ubuntu di Linux, ma può darsi che non sia la versione più recente. Per installare Python 3 su Ubuntu procedete così:

1. Fate clic sul pulsante dell’Ubuntu Software Center nella Sidebar (è l’icona che ha l’aspetto di una borsa arancione – se non la vedete, potete sempre fare clic sull’icona Dash Home e inserire *Software* nella finestra di dialogo).
2. Inserite *Python* nella casella di ricerca (in alto al centro) del Software Center.
3. Nell’elenco del software che si presenta, selezionate la versione più recente di IDLE, che è, al momento in cui scrivo, *IDLE (using Python 3.5)*.
4. Fate clic su **Installa**.
5. Inserite la password di amministratore quando vi viene richiesta e procedete all’autenticazione. (Non conoscete la password di amministratore? Dovrete chiedere ai vostri genitori di inserirla per voi.)

NOTA

In qualche versione di Ubuntu, può darsi che vediate solo Python (v3.5) nel menu principale (anziché IDLE): potete installare quello.

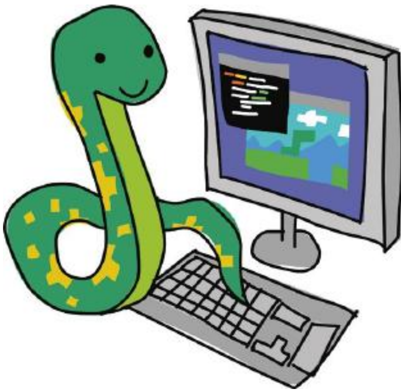
Con alcune versioni di Ubuntu la ricerca di Python attraverso il Software Center sembra non dare frutto. In tal caso, aprite una finestra di Terminale e al prompt inserite il comando seguente:

```
sudo apt-get install idle-python3.5
```

L’installazione è poi completamente automatica, tranne una richiesta di conferma (a cui dovete rispondere con un Sì). Al termine potrete trovare *IDLE (using Python 3.5)* nell’elenco del software installato nel Software Center, nella sezione Applicazioni. Se avete difficoltà a trovarlo, utilizzate la casella di ricerca di *Cerca sul computer* (è la prima icona in alto nella barra laterale).

DOPO AVER INSTALLATO PYTHON

Ora sul desktop del vostro computer (se avete una macchina Windows o un Mac OS X) dovreste avere un'icona **IDLE**. Se usate Ubuntu, a seconda della versione del sistema operativo, troverete l'icona nel menu **Applicazioni** oppure nel Software Center.



Fate un doppio clic sull'icona o scegliete l'opzione da menu e vi comparirà una finestra come questa:

```
Python 3.5.1 Shell
File  Edit  Shell  Debug  Options  Window  Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> |
                                                                Ln: 3  Col: 4
```

Questa è la *shell di Python*, che fa parte dell'ambiente di sviluppo integrato. I tre simboli "maggiore di" (>>>) sono il cosiddetto *prompt*.

Inseriamo qualche comando al prompt, cominciando con questo:

```
>>> print("Ciao mondo")
```

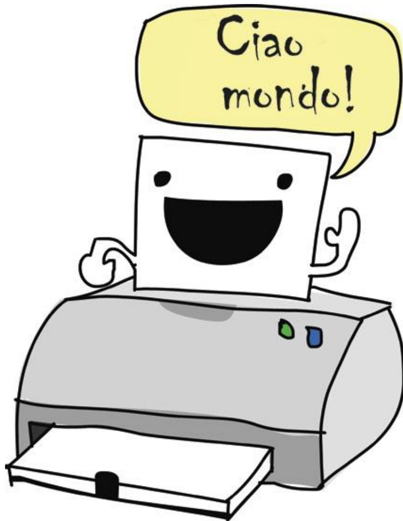
Fate attenzione a non dimenticare i doppi apici (" "). Premete Invio sulla tastiera, quando avete finito di scrivere questa riga.

Se avete inserito correttamente il comando, vedrete qualcosa di analogo a questo:

```
>>> print("Ciao Mondo")
Ciao Mondo
>>>
```

Sarà ricomparso il prompt, per farvi sapere che la shell di Python è pronta ad accettare altri comandi.

Congratulazioni! Avete appena creato il vostro primo programma in Python. La parola `print` fa parte di quel tipo di comandi Python che sono chiamati *funzioni*, e "stampa" qualsiasi cosa si trovi all'interno delle parentesi.



In sostanza, avete dato al computer l'istruzione di visualizzare le parole "Ciao mondo" – una istruzione comprensibile sia a voi che al computer.

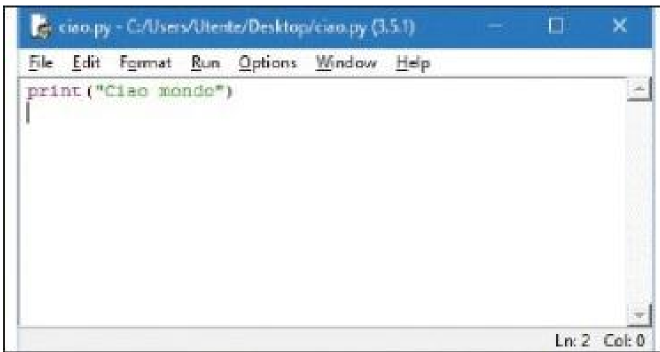
SALVARE I PROGRAMMI PYTHON

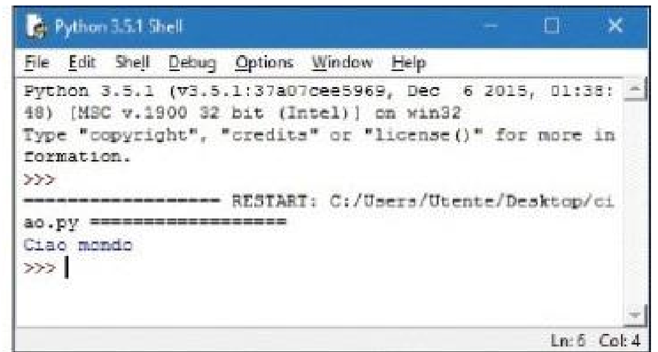
I programmi Python non sarebbero molto utili se fosse necessario riscriverli ogni volta che li si vuole usare, oppure visualizzare in modo da poterli consultare. Certo, un programma molto breve si potrebbe anche riscrivere, ma un programma di grandi dimensioni, come un elaboratore di testo, può essere costituito da milioni di righe di codice. Stampatele tutte e vi ritroverete magari con un malloppo da oltre 100.000 pagine. Difficile portarlo a casa sotto braccio – sperando che non arrivi una folata di vento.

Per fortuna, è possibile salvare i programmi per poterli riusare in futuro. Per salvare un nuovo programma, in IDLE scegliete **File, New File**. Si aprirà una nuova finestra, con il nome **Untitled** nella barra del titolo. Nella nuova finestra della shell scrivete questo codice:

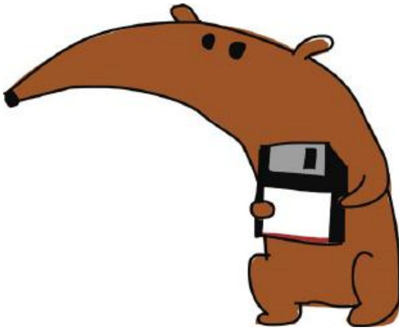
```
print("Ciao mondo")
```

Ora scegliete **File**, **Save**. Quando viene chiesto il nome del file, scrivete *ciao.py* e salvate il file sul desktop. Poi scegliete **Run**, **Run Module**. Se non avete commesso errori, il programma verrà eseguito, con questo risultato:



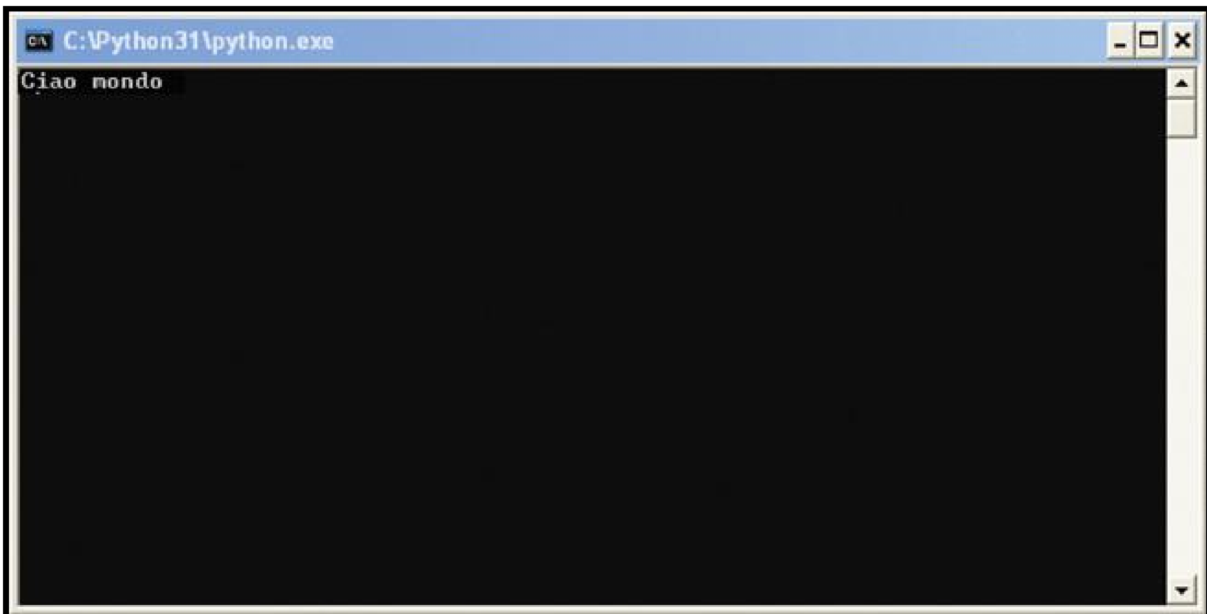


Ora, se chiudete la finestra della shell ma lasciate aperta la finestra ciao.py e scegliete di nuovo **Run, Run Module**, ricomparirà la shell di Python e il vostro programma verrà nuovamente eseguito. (Per riaprire la shell di Python senza eseguire il programma, scegliete **Run, Python Shell**.



Dopo aver eseguito il codice, sul desktop troverete una nuova icona, con il nome *hello.py*. Se fate un doppio clic su quell'icona, apparirà brevemente una finestra con lo sfondo nero, e subito dopo svanirà. Che cosa è successo?

Avete visto la console da riga di comando di Python (analoga alla shell) che si avviava, stampava "Ciao mondo" e poi si chiudeva. Ecco quello che vi apparirebbe, se aveste una vista ultrarapida da supereroe e riusciste a vedere la finestra prima che si chiuda:



Oltre ai menu, potete usare i tasti di scelta rapida per creare una nuova finestra della shell, salvare un file ed eseguire un programma:

- In Windows e Ubuntu, usate Ctrl-N per creare una nuova finestra della shell, Ctrl-S per salvare il file dopo che avete finito di scriverlo, F5 per eseguire il programma.
- In Mac OS X, usate Opzione-N per creare una nuova finestra della shell, Opzione-S per salvare il file, mentre per eseguire il programma premete e tenuto premuto il tasto funzione (FN) e premete F5.

CHE COSA AVETE IMPARATO

Abbiamo iniziato in questo capitolo con un'applicazione Ciao mondo, il programma con cui praticamente tutti iniziano quando cominciano a studiare la programmazione. Nel prossimo capitolo, faremo qualche altra cosa utile con la shell di Python.

2
CALCOLI E VARIABILI

Ora che avete installato Python e sapete come avviare la shell, siete pronti per fare qualcosa. Inizieremo con qualche semplice calcolo e poi passeremo alle variabili.
Le *variabili* sono un modo per conservare le cose in un programma e permettono di scrivere programmi utili.

CALCOLARE CON PYTHON

Normalmente, se vi chiedessero di calcolare il prodotto di due numeri, per esempio 8 × 3.57 (useremo il punto al posto della virgola, nei numeri decimali, come nell'uso anglosassone: è così che li vuole scritti Python), usereste una calcolatrice o carta e penna. Beh, che ne dite di usare la shell di Python per fare questo calcolo? Proviamo.

Avviate la shell di Python con un doppio clic sull'icona IDLE sul desktop o, se usate Ubuntu, facendo clic sull'icona IDLE nel Software Center. Al prompt, inserite l'espressione:

```
>>> 8 * 3.57
28.56
```

Notate che, per chiedere a Python di effettuare una moltiplicazione, si usa il simbolo dell'asterisco (*), invece del tradizionale segno "per" (×).

E se provassimo con un calcolo un po' più utile?

Supponiamo che, scavando in giardino, troviate un sacchetto con 20 monete d'oro. Il giorno dopo, vi intrufolate in cantina e infilate le monete nella macchina replicatrice a vapore del nonno (per fortuna, riuscite giusto a infilarci tutte le venti monete). Sentite un risucchio e uno schiocco e, dopo qualche ora, ecco che ne escono altre 10 monete luccicanti.

Quante monete avreste nella cassa del vostro tesoro se faceste la stessa cosa ogni giorno per un anno? Sulla carta, le espressioni potrebbero essere queste:

10 × 365 = 3650

20 + 3650 = 3670

Certo, è facile fare questi calcoli con la calcolatrice o su carta, ma possiamo farli anche con la shell di Python. Prima, moltiplichiamo le 10 monete per i 365 giorni in un anno, poi al risultato sommiamo le 20 monete originali.

```
>>> 10 * 365
3650
>>> 20 + 3650
3670
```

Un corvo vede le belle monete d'oro nella vostra camera da letto e ogni settimana entra e riesce a portarsene via tre. Quante monete vi rimarranno alla fine dell'anno? Ecco come si presenterebbe il calcolo nella shell:

```
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Prima, moltiplichiamo 3 monete per 52 settimane. Il risultato è 156. Sottraiamo questo numero dal nostro totale (3670) e scopriamo che a fine anno vi rimarranno 3514 monete.

Questo è un programma molto semplice. Nel corso del libro, vedrete come sviluppare queste idee per scrivere programmi molto più utili.

GLI OPERATORI DI PYTHON

Nella shell di Python potete eseguire le quattro operazioni aritmetiche fondamentali (somma, sottrazione, moltiplicazione e divisione), oltre a molte altre che per il momento non affrontiamo. I simboli fondamentali utilizzati da Python per le operazioni matematiche sono chiamati *operatori* e sono elencati nella Tabella 2.1.

Tabella 2.1. Gli operatori fondamentali di Python.

Simbolo	Operazione
+	Somma
-	Sottrazione
*	Moltiplicazione
/	Divisione

Si usa la *barra per la divisione* perché è simile alla linea che si scrive per indicare le frazioni. Per esempio, se avete 100 pirati e 10 grandi botti e volete calcolare quanti pirati dovete sistemare in ciascuna botte, potete dividere 100 pirati per 20 botti (100 : 20) scrivendo `100 / 20` nella shell di Python.

L'ORDINE DELLE OPERAZIONI

In un linguaggio di programmazione si usano le parentesi per controllare l'ordine di esecuzione delle operazioni. Una *operazione* è un'istruzione che usa un operatore. La moltiplicazione e la divisione hanno la precedenza su somma e sottrazione, cioè vengono eseguite per prime. In altre parole, se inserite

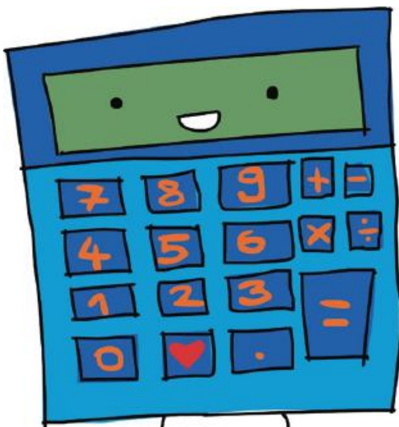
un’espressione in Python, le moltiplicazioni e le divisioni vengono eseguite prima delle somme e delle sottrazioni. Per esempio, in questo caso, prima vengono moltiplicati 30 e 20, poi al prodotto viene sommato 5.

```
>>> 5 + 30 * 20
605
```

Questa espressione è il nostro modo di dire “moltiplica 30 per 20, poi somma 5 al risultato”. Il totale è 605. Possiamo modificare l’ordine in cui vengono eseguite le operazioni inserendo le parentesi, per esempio così, per sommare 5 e 30 prima che venga eseguita la moltiplicazione per 20:

```
>>> (5 + 30) * 20
700
```

Il risultato di questo calcolo è 700 (non 605) perché le parentesi dicono a Python di calcolare prima la somma fra parentesi e solo dopo eseguire l’operazione indicata fuori dalle parentesi. L’esempio dice “somma 5 a 30, poi moltiplica il risultato per 20”.



Le parentesi possono essere annidate, cioè possono esserci parentesi dentro parentesi, come in questo caso:

```
>>> ((5 + 30) * 20) / 10
70.0
```

In questo caso, Python svolge prima l’operazione indicata dalle parentesi più interne, poi quella dentro le parentesi esterne, infine quella fuori dalle parentesi (la divisione).

In altre parole, questa espressione dice “somma 5 a 30, poi moltiplica il risultato per 20 e dividi il nuovo risultato per 10”. Ecco che cosa succede:

- Sommando 5 e 30 si ottiene 35.
- Moltiplicando 35 per 20 si ottiene 700.
- Dividendo 700 per 10 si ottiene il risultato finale, 70.

Se non avessimo usato le parentesi, il risultato sarebbe stato leggermente

diverso:

```
>>> 5 + 30 * 20 / 10
65.0
```

In questo caso, prima 30 viene moltiplicato per 20 (risultato: 600), poi 600 viene diviso per 10 (risultato: 60) e infine viene sommato 5, il che dà come risultato finale 65.

ATTENTI

Ricordate che moltiplicazione e divisione vengono eseguite prima di somma e sottrazione, a meno che vengano usate le parentesi per controllare l'ordine di esecuzione delle operazioni.

LE VARIABILI SONO COME ETICHETTE

La parola *variabile* nella programmazione descrive un luogo in cui conservare informazioni come numeri, testo, elenchi di numeri e testi e così via. Si può considerare una variabile anche come un'etichetta che indica qualcosa.

Per esempio, per creare una variabile che si chiama `fred`, possiamo usare un segno di uguale `(=)` e poi dire a Python per che tipo di informazione quella variabile debba essere l'etichetta. Qui creiamo la variabile `fred` e diciamo a Python che è l'etichetta del numero 100 (notate che questo non esclude che anche un'altra variabile possa avere lo stesso valore):

```
>>> fred = 100
```

Per scoprire di quale valore una variabile sia l'etichetta, nella shell scrivete `print`, seguito dal nome della variabile fra parentesi, in questo modo:

```
>>> print(fred)
100
```

Possiamo anche dire a Python di modificare la variabile `fred` in modo che indichi qualcos'altro. Per esempio, ecco come cambiare il valore di `fred` in 200:

```
>>> fred = 200
>>> print(fred)
200
```

Nella prima riga, diciamo che `fred` è un'etichetta del numero 200. Nella seconda, chiediamo che cosa etichetta `fred`, tanto per essere sicuri della variazione. Python stampa il risultato sull'ultima riga.

Possiamo usare anche più di un'etichetta (più di una variabile) per indicare la stessa cosa:

```
>>> fred = 200
>>> john = fred
>>> print (john)
200
```

In questo esempio, diciamo a Python che vogliamo che il nome (la variabile) `john` etichetti la stessa cosa di `fred` utilizzando il segno di uguale fra `john` e `fred`.

Ovviamente, `fred` probabilmente non è un nome molto utile per una variabile, perché non ci dice nulla di quello per cui quella variabile viene utilizzata. Chiamiamo allora la nostra variabile `numero_di_monete` anziché `fred`, così:

```
>>> numero_di_monete = 200
>>> print(numero_di_monete)
200
```

In questo modo è chiaro che parliamo di 200 monete.

I nomi delle variabili possono essere costituiti da lettere, numeri e dal carattere underscore (`_`), ma non possono iniziare con un numero. Potete usare anche una singola lettera (per esempio `a`), così come una lunga frase, come nome di variabile. (Una variabile non può contenere spazi, perciò usate l'underscore per separare le parole.) A volte, se dovete fare qualcosa rapidamente, nomi brevi per le variabili sono la soluzione migliore. Il nome che sceglìete dipende da quanto volete che sia significativo.

Ora che sapete come creare variabili, vediamo in che modo usarle.

USARE LE VARIABILI

Ricordate la nostra espressione per calcolare quante monete avreste avuto alla fine dell'anno, potendo magicamente creare nuove monete con la folle invenzione del nonno in cantina? Abbiamo queste espressioni:

```
>>> 20 + 10 * 365
3670
>>> 3 * 52
156
>>> 3670 - 156
3514
```

Possiamo trasformarle in un'unica riga di codice:

```
>>> 20 + 10 * 365 - 3 * 52
3514
```

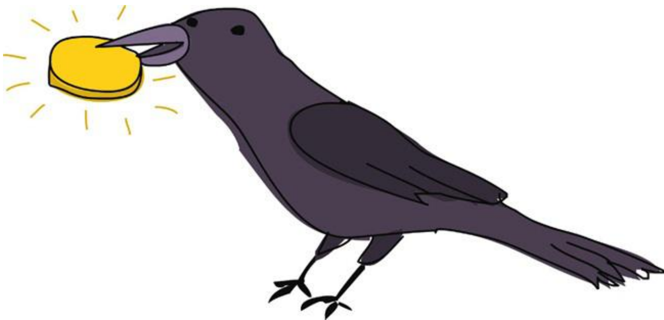
Ora, che ne dite di trasformare i numeri in variabili? Provate:

```
>>> monete_trovate = 20
>>> monete_magiche = 10
>>> monete_rubate = 3
```

Queste righe creano le variabili `monete_trovate`, `monete_magiche` e `monete_rubate`. Ora possiamo riscrivere la nostra espressione in questo modo:

```
>>> monete_trovate + monete_magiche * 365 - monete_rubate * 52
3514
```

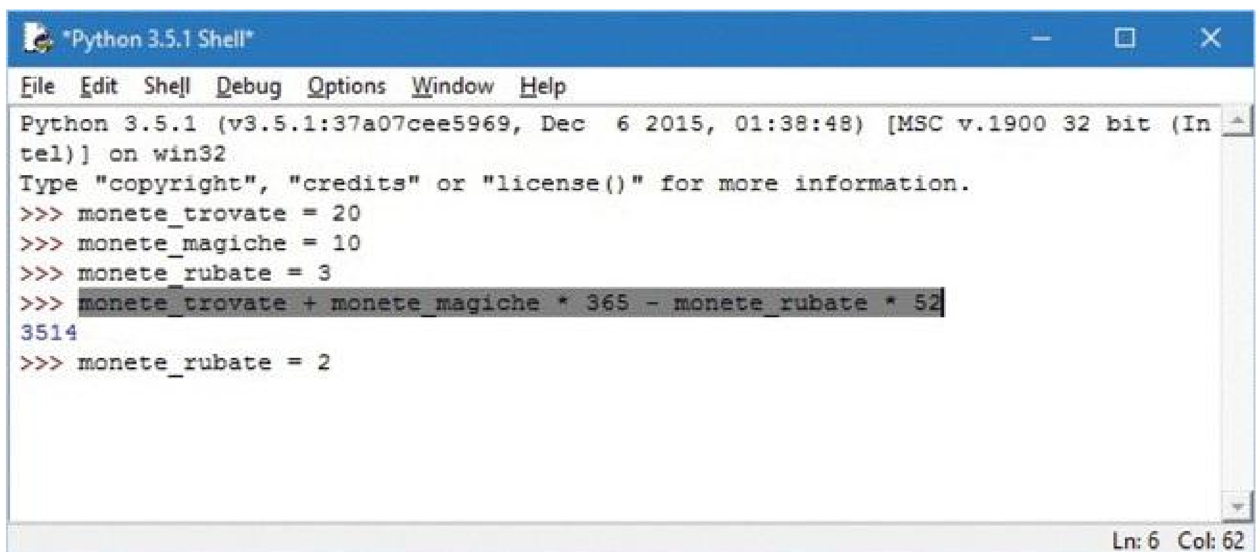
Come si può vedere, si ottiene sempre la stessa risposta. E allora, perché darsi tanto da fare? Ah, ma qui sta la magia delle variabili. Che cosa succede se mettete uno spaventapasseri alla finestra e il corvo ruba solo due monete invece di tre? Se usiamo una variabile, possiamo semplicemente modificarla in modo che contenga il nuovo numero, e così il cambiamento si comunicherà in tutti i punti in cui quel numero era usato nell'espressione. Possiamo cambiare la variabile `monete_rubate` in questo modo:



```
>>> monete_rubate = 2
```

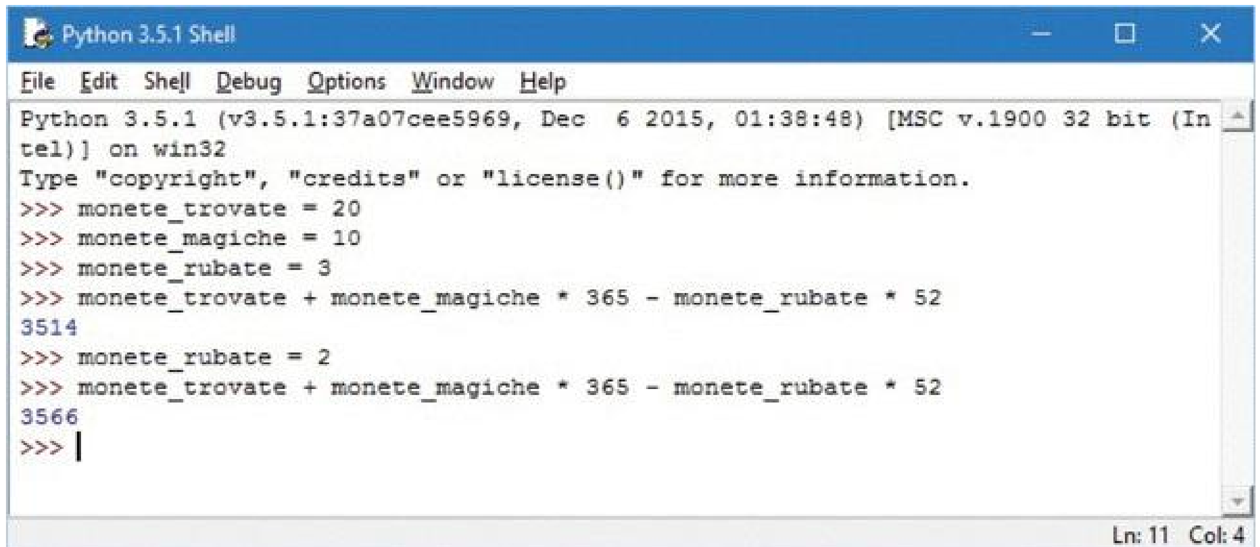
Poi possiamo copiare e incollare l'espressione, per calcolare nuovamente il totale, in questo modo:

1. Selezionate il testo da copiare facendo clic con il mouse e poi trascinando dall'inizio alla fine della riga:



2. Tenete premuto il tasto Ctrl (o il tasto Opzione, se usate un Mac) e premete C per copiare il testo selezionato. (D'ora in poi, scriveremo semplicemente Ctrl-C.)

3. Fate clic sull’ultima riga di prompt (dopo `monete_rubate = 2`).
4. Tenete premuto il tasto Ctrl e premete V per incollare il testo selezionato. (D’ora in poi scriveremo semplicemente Ctrl-V).
5. Premete Invio per vedere il nuovo risultato.



Non è molto più facile che riscrivere tutta l’espressione?

Potete provare a modificare le altre variabili, poi copiate (Ctrl-C) e incollate (Ctrl-V) l’espressione per vedere l’effetto dei cambiamenti. Per esempio, se assestando un colpo al momento opportuno sul fianco dell’invenzione del nonno questa emette ogni volta 3 monete in più, vi ritroverete alla fine dell’anno con 4661 monete:

```
>>> monete_magiche = 13
>>> monete_trovate + monete_magiche * 365 - monete_rubate * 52
4661
```

Ovviamente, usare le variabili per un’espressione semplice come questa non è di *grande* utilità: non siamo ancora arrivati a qualcosa di *veramente* utile. Per il momento, ricordate solo che le variabili sono un modo per etichettare cose, in modo da poterle riutilizzare in seguito.

CHE COSA AVETE IMPARATO

In questo capitolo avete imparato come calcolare semplici espressioni matematiche con gli operatori di Python e come usare le parentesi per controllare l’ordine di esecuzione delle operazioni (cioè l’ordine in cui Python calcola le diverse parti di un’espressione). Poi abbiamo creato delle variabili per etichettare dei valori e abbiamo usato quelle variabili nei nostri calcoli.

3
STRINGHE, LISTE, TUPLE E MAPPE

Nel Capitolo 2, abbiamo eseguito qualche semplice calcolo con Python, e avete scoperto che cosa sono le variabili. In questo capitolo, lavoreremo con alcuni degli altri elementi nei programmi Python: stringhe, liste, tuple e mappe. Userete le stringhe per presentare dei messaggi nei vostri programmi (per esempio “Preparatevi” o “Game over” in un gioco). Scoprirete anche come liste, tuple e mappe vengano utilizzate per conservare insiemi di cose.

STRINGHE

Nella programmazione, si dice *stringa* una successione di caratteri. Una stringa può contenere tutte le lettere, i numeri e i simboli di questo libro, oppure il vostro nome e indirizzo. In effetti, il primo programma Python creato nel Capitolo 1 usava una stringa: “Ciao mondo”.

CREARE STRINGHE

In Python, si crea una stringa mettendo doppi apici all’inizio e alla fine di un testo, perché i linguaggi di programmazione devono distinguere tipi di valori diversi. (Dobbiamo dire se un valore è un numero, una stringa o altro.) Per esempio, prendiamo la variabile `fred` del Capitolo 2 e usiamola per etichettare una stringa:

```
fred = "Perché i gorilla hanno narici grandi? Dita grandi!!"
```

Poi, per vedere che cosa c’è dentro `fred`, possiamo scrivere:

```
>>> print (fred)
Perché i gorilla hanno narici grandi? Dita grandi!!
```

Si possono usare anche apici singoli, per creare una stringa, in questo modo:

```
>>> fred = 'Trentatre trentini entrarono in Trento'
>>> print(fred)
Trentatre trentini entrarono in Trento
```

Però, se cercate di inserire più righe di testo per una stringa usando solo un apice singolo (') o doppio (") o se iniziate con un tipo di apice e finite con l’altro, la shell di Python vi mosterà un messaggio di errore. Per esempio, provate a inserire questa riga:

```
>>> fred = “Come fanno i dinosauri a pagare le bollette?
```

Vedrete questo risultato:

```
SyntaxError: EOL while scanning string literal
```

La scritta è un messaggio di errore che lamenta un errore di sintassi, perché non avete osservato la regola, che prescrive la chiusura di una stringa con un apice singolo o doppio. [La scritta significa: Errore di sintassi: raggiunta la fine della riga nell'analizzare la stringa.]

Sintassi significa il modo in cui sono disposte e ordinate le parole in una frase o, in questo caso, il modo in cui sono disposti e ordinati parole e simboli in un programma. `SyntaxError` quindi significa che avete fatto qualcosa in un ordine che Python non si aspettava, o che Python si aspettava qualcosa che invece mancava. *EOL* significa *end-of-line* (fine riga), perciò il resto del messaggio di errore dice che Python ha raggiunto la fine della riga e non ha trovato un doppio apice di chiusura della stringa.

Per usare più di una riga di testo nella stringa (che si chiama allora *stringa multilinea*), iniziate la stringa con tre apici singoli (''') e premete `Invio` fra una riga e l'altra, così:

```
>>> fred = '''Come fanno i dinosauri a pagare le bollette?
Con assegni da tirannosauro!'''
```

Proviamo ora a stampare i contenuti di `fred` per vedere se è andato tutto bene:

```
>>> print(fred)
Come fanno i dinosauri a pagare le bollette?
Con assegni da tirannosauro!
```

TRATTARE PROBLEMI CON LE STRINGHE

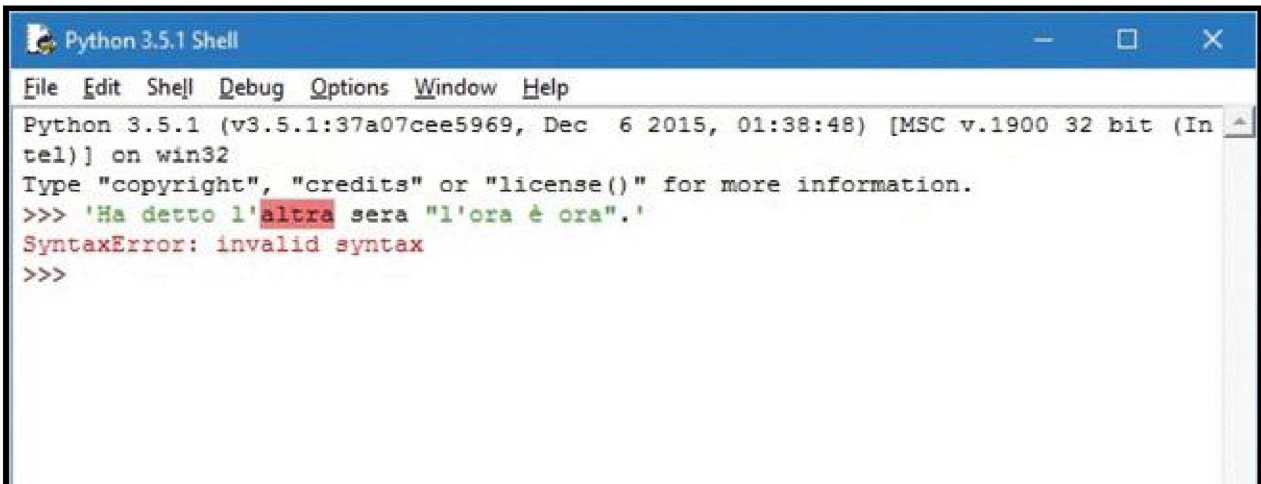
Prendiamo questo esempio di stringa, che provoca un messaggio di errore in Python:

```
>>> stringa = 'Ha detto l'altra sera: "l'ora è ora".'
SyntaxError: invalid syntax
```

Nella prima riga, abbiamo cercato di creare una stringa (definita con il nome molto fantasioso di `stringa`), racchiusa fra apici singoli, ma contenente una combinazione di apici e anche di doppi apici. Un bel pasticcio!

Ricordate che Python non è intelligente quanto un essere umano, perciò tutto quello che vede è una stringa contenente `Ha detto l'`, seguito da una serie di altri caratteri che non si aspettava. Quando Python vede un apice (singolo o doppio) si aspetta una stringa che inizia con il primo apice (singolo o doppio) e che si conclude con il successivo apice corrispondente. Qui, l'inizio della stringa è indicato dal primo apice singolo prima di `Ha`, e la fine della stringa, per quel che consta a Python, è l'apice singolo dopo la `l` di `l'altra sera`. IDLE evidenzia dove le

cose iniziano a non quadrare:

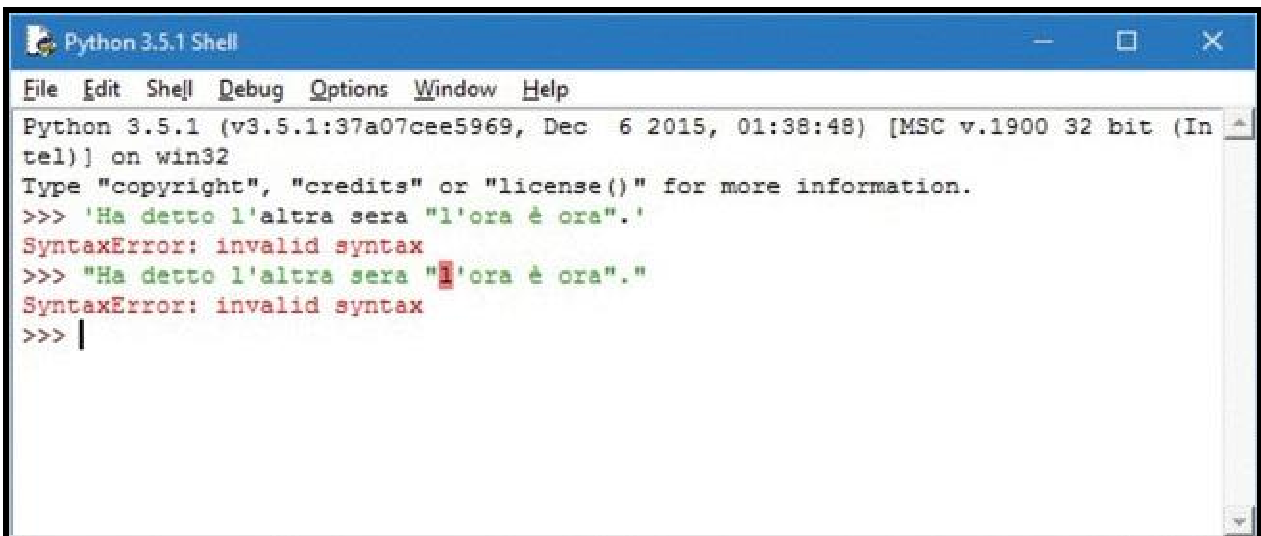


L’ultima riga di IDLE ci dice che tipo di errore si è verificato: in questo caso un errore di sintassi non valida.

L’uso di apici doppi al posto di quelli singoli produce ancora un errore:

```
>>> stringa = "Ha detto l'altra sera "l'ora è ora"."
SyntaxError: invalid syntax
```

Qui, Python vede una stringa delimitata da apici doppi e interpreta correttamente gli apici singoli come caratteri (e non come delimitatori), ma poi, dopo `sera`, incontra un doppio apice che interpreta come fine della stringa – seguito da altri caratteri che non sa più come interpretare:



Tutto questo perché, dal punto di vista di Python, tutti quei caratteri ulteriori semplicemente non dovrebbero esserci. Python cerca il tipo di apice corrispondente a quello di apertura e, quando lo incontra, pensa che la stringa sia finita e non sa che cosa farsene di quello che segue sulla stessa riga.

La soluzione è una stringa multilinea, come abbiamo visto, con l’uso di tre apici singoli ('''), che permettono di combinare apici singoli e doppi nella stringa senza provocare errori. In effetti, se si usano come delimitatori i tre apici singoli, nella

stringa si può mettere qualsiasi combinazione di apici singoli e doppi (basta non tentare di inserire tre apici singoli in fila). Questa è la versione priva di errori della nostra stringa:

```
stringa = '''Ha detto l'altra sera "l'ora è ora".'''
```

Ma aspettate, non è finita. Se volete proprio usare un apice singolo o doppio per delimitare una stringa in Python, invece di usare i tre apici singoli, potete aggiungere una barra rovesciata (\) davanti a ciascun apice all'interno della stringa. Questa tecnica si chiama *escaping* e la barra rovesciata è chiamata *carattere di escape*. È un modo per dire a Python "Sì, lo so, ho degli apici dentro la mia stringa, e voglio che tu li ignori finché non vedi l'apice finale".

Le stringhe con i caratteri di escape possono essere più difficili da leggere, perciò è meglio, in generale, usare le stringhe multilinea. Può capitare però di imbattersi in qualche frammento di codice che usa i caratteri di escape, perciò è bene sapere che cosa sono quelle barre rovesciate.

Ecco qualche esempio della tecnica dell'escaping:

```
❶ >>> apice_singolo = 'Ha detto l\'altra sera "l\'ora è ora".'
❷ >>> doppio_apice = "Ha detto l'altra sera \"l'ora è ora\"."
>>> print(apice_singolo)
Ha detto l'altra sera "l'ora è ora".
>>> print(doppio_apice)
Ha detto l'altra sera "l'ora è ora".
```

Prima, in ❶, abbiamo creato una stringa con l'apice singolo come delimitatore, mettendo la barra rovesciata di fronte agli apici singoli all'interno della stringa. In ❷, abbiamo creato una stringa con il doppio apice e usato la barra rovesciata davanti ai doppi apici dentro la stringa. Nelle righe successive, abbiamo stampato le variabili appena create. Notate che il carattere della barra rovesciata non compare nelle stringhe, quando vengono stampate.

INCORPORARE VALORI NELLE STRINGHE

Se volete visualizzare un messaggio che usi i contenuti di una variabile, potete incorporare i valori in una stringa con `%s`, che è una sorta di indicatore di un valore da aggiungere in seguito (si parla di *incorporare valori* o di *sostituzione di stringa*, nel gergo dei programmatori). Per esempio, perché Python calcoli o ricordi il numero di punti fatti in un gioco e poi li inserisca in una frase come "Ho totalizzato ___ punti", usate `%s` nella frase al posto di quel valore, poi dite a Python il valore, per esempio così:

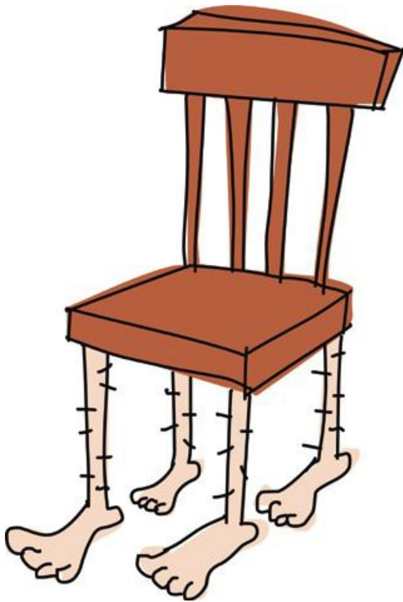
```
>>> miopunteggio = 1000
>>> messaggio = 'Ho totalizzato %s punti'
>>> print(messaggio % miopunteggio)
Ho totalizzato 1000 punti
```

Abbiamo creato la variabile `miopunteggio` con valore `1000` e la variabile `messaggio` con una stringa che contiene le parole “Ho totalizzato `%` punti”, dove `%s` è un “segnaposto” per il numero di punti, Nella riga successiva, chiamiamo `print(messaggio)` con il simbolo `%` per dire a Python di sostituire `%s` con il valore della variabile `miopunteggio`. Il risultato è la visualizzazione del messaggio `Ho totalizzato 1000 punti`. Non è necessario usare una variabile per il valore: avremmo ottenuto lo stesso risultato utilizzando (`messaggio % 1000`).

Possiamo anche passare valori diversi per il segnaposto `%s`, utilizzando variabili diverse, come in questo esempio:

```
>>> battuta = '%s: uno strumento per trovare i mobili al buio'
>>> parte_del_corpo1 = 'Ginocchio'
>>> parte_del_corpo2 = 'Stinco'
>>> print(battuta % parte_del_corpo1)
Ginocchio: uno strumento per trovare i mobili al buio
>>> print(battuta % parte_del_corpo2)
Stinco: uno strumento per trovare i mobili al buio
```

Qui abbiamo creato tre variabili. La prima, `battuta`, conserva la stringa con il segnaposto `%s`. Le altre due variabili sono `parte_del_corpo1` e `parte_del_corpo2`. Possiamo stampare la variabile `battuta` e usare nuovamente l’operatore `%` che verrà sostituito dai contenuti delle variabili `parte_del_corpo1` e `parte_del_corpo2` per generare messaggi diversi.



Potete usare anche più di un segnaposto in una stringa, per esempio in questo modo:

```
>>> numeri = 'Che cosa ha detto il numero %s al numero %s? Bella
cintura!'
>>> print(numeri % (0, 8))
Che cosa ha detto il numero 0 al numero 8? Bella cintura!
```

Quando si usa più di un segnaposto, bisogna ricordare di inserire i valori

sostitutivi fra parentesi, come nell'esempio. L'ordine in cui i valori compaiono fra le parentesi sarà anche l'ordine in cui verranno usati nella stringa.

MOLTIPLICARE STRINGHE

Quanto fa 10 moltiplicato per 5? La risposta, ovviamente, è 50. Ma che cosa fa 10 moltiplicato per a? Questa è la risposta di Python:

```
>>> print(10 * 'a')
Aaaaaaaaaa
```

Programmando in Python si può usare questo metodo per allineare le stringhe con un numero specifico di spazi, quando si visualizzano messaggi nella shell, per esempio. Potremmo stampare una lettera nella shell: selezionate **File > New File** e inserite questo codice:

```
spazi = ' '* 25
print('%s Dottor Morelli' % spazi)
print('%s Via Foresta Nera 33' % spazi)
print('%s Costa Sbadiglio' % spazi)
print()
print()
print('Egregio Signore')
print()
print ('Vorrei segnalare che mancano delle tegole dal tetto')
print ('del gabinetto esterno.')
print ('Penso che siano volate via per il vento di lunedì.')
print ()
print ('Ossequi')
print ('Merlino Veggenti')
```

Scritto il codice nella finestra della shell, selezionate **File > Save As**. Date al file il nome *mialettera.py*.
Potete poi eseguirlo selezionando **Run > Run Module**.

NOTA

Da questo momento, quando vedrete Save As: unnomedifile.py *al di sopra di un brano di codice, saprete che dovete selezionare* **File > New File**, *inserire il codice nella finestra che si apre e poi salvarlo come abbiamo fatto in questo esempio.*

Nella prima riga di questo esempio abbiamo creato la variabile `spazi` assegnandole come valore un carattere spazio moltiplicato per 25. Poi abbiamo usato quella variabile, nelle tre righe successive, per allineare il testo rientrato sulla destra della finestra della shell. Potete vedere il risultato di questi enunciati `print` qui sotto:

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
                          Dottor Morelli
                          Via Foresta Nera 33
                          Costa Sbadiglio


Egregio Signore

Vorrei segnalare che mancano delle tegole dal tetto
del gabinetto esterno.
Penso che siano volate via per il vento di lunedì.

Ossequi
Merlino Veggenti
>>> 
                                                  Ln: 18 Col: 4
```

Oltre a usarla per l’allineamento, possiamo usare la moltiplicazione per riempire lo schermo di messaggi irritanti. Potete provare da soli questo esempio.

```
>>> print(1000 * 'snirt')
```

LE LISTE SONO PIÙ POTENTI DELLE STRINGHE

“Zampe di ragno, dito di rana, occhio di girino, ala di pipistrello, bava di lumacone e forfora di serpente”: non è proprio una lista della spesa normale (a meno che non siate maghi), ma la useremo come primo esempio per illustrare le differenze fra stringhe e liste.

Potremmo conservare questa lista nella variabile `mago_lista` con una stringa come questa:

```
>>> mago_lista = 'zampe di ragno, dito di rana, occhio di girino,
ala di pipistrello, bava di lumacone, forfora di serpente'
>>> print(mago_lista)
zampe di ragno, dito di rana, occhio di girino, ala di
pipistrello, bava di lumacone, forfora di serpente
```

Potremmo però anche creare una lista, un tipo di oggetto un po’ magico di Python, che si può manipolare meglio. Ecco come si presenterebbero i nostri ingredienti in forma di lista:

```
>>> mago_lista = [ìzampe di ragnoì, ìdito di ranaì, 'occhio di
girino', 'ala di pipistrello', 'bava di lumacone', 'forfora di
serpente']
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'occhio di girino', 'ala di
pipistrello', 'bava di lumacone', 'forfora di serpente']
```

Per creare una lista bisogna scrivere un po’ di caratteri in più che per creare una stringa, ma una lista è più utile perché può essere manipolata. Per esempio, potremmo stampare il terzo elemento della `mago_lista` (occhio di girino) indicando la sua posizione nella lista (il suo *indice*) fra parentesi quadre (`[]`), in questo modo:

```
>>> print(mago_lista[2])
occhio di girino
```

Cosa? Ma non era il terzo elemento della lista? Vero, ma le liste iniziano con la posizione 0, perciò il primo elemento di una lista è 0, il secondo 1, il terzo 2. Forse non avrà molto senso per gli esseri umani, ma ne ha per i computer.

Possiamo anche modificare un elemento in una lista molto più facilmente che in una stringa. Magari, anziché un occhio di girino avevamo bisogno di una lingua di chiocciola. Ecco come potremmo fare con la nostra lista:

```
>>> mago_lista[2] = 'lingua di chiocciola'
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'lingua di chiocciola', 'ala
di pipistrello', 'bava di lumacone', 'forfora di serpente']
```

In questo modo abbiamo modificato l’elemento di indice 2: prima era “occhio di girino”, ora è “lingua di chiocciola”.

Altra possibilità è visualizzare un sottoinsieme degli elementi nella lista. Lo si può fare usando un segno di due punti (`:`) dentro le parentesi quadre. Per esempio, per vedere gli elementi dal terzo al quinto della lista:

```
>>> print(mago_lista[2:5])
['lingua di chiocciola', 'ala di pipistrello', 'bava di
lumacone']
```

Scrivere [2:5] equivale a dire “mostrami gli elementi dalla posizione di indice 2 fino a quella (esclusa) di indice 5” – in altre parole, gli elementi 2, 3 e 4.

Le liste possono essere utilizzate per conservare elementi di qualsiasi genere, per esempio numeri:

```
>>> qualche_numero = [1, 2, 5, 10, 20]
```

Possono contenere anche stringhe:

```
>>> qualche_stringa = ['Quale', 'strega', 'è', 'cattiva']
```

Possono contenere anche numeri e stringhe alla rinfusa:

```
>>> numeri_e_stringhe = ['Perché', 'avevo', 6, 'paura', 'di', 7,
"per colpa", 7, 8, 9]
>>> print(numeri_e_stringhe)
['Perché', 'avevo', 6, 'paura', 'di', 7, 'per colpa', 7, 8, 9]
```

Le liste poi possono contenere anche altre liste:

```
>>> numeri = [1, 2, 3, 4]
>>> stringhe = ['Ho', 'pestato', 'il', 'ditone', 'e', 'adesso',
'mi', 'fa', 'male']
>>> mialista = [numeri, stringhe]
>>> print(mialista)
[[1, 2, 3, 4], ['Ho', 'pestato', 'il', 'ditone', 'e', 'adesso',
'mi', 'fa', 'male']]
```

In quest’ultimo esempio, abbiamo creato tre variabili: `numeri` che contiene quattro numeri, `stringhe` che contiene nove stringhe e `mialista` utilizzando numeri e stringhe. La terza lista (`mialista`) ha solo due elementi, perché è una lista di nomi di variabili, non dei contenuti delle variabili stesse.

AGGIUNGERE ELEMENTI A UNA LISTA

Per aggiungere elementi a una lista, si usa la funzione `append`. Una funzione è un blocco di codice che dice a Python di fare qualcosa. In questo caso, `append` aggiunge

un elemento alla fine di una lista.

Per esempio, per aggiungere un rutto di orso (sono sicuro che esiste una cosa del genere) alla lista della spesa del mago:

```
>>> mago_lista.append('rutto di orso')
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'lingua di chiocciola', 'ala
di pipistrello', 'bava di lumacone', 'forfora di serpente',
'rutto di orso']
```

Potete continuare ad aggiungere elementi magici alla lista, sempre nello stesso modo:

```
>>> mago_lista.append('mandragora')
>>> mago_lista.append('cicuta')
>>> mago_lista.append('gas di palude')
```

Ora la lista del mago sarà diventata questa:

```
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'lingua di chiocciola', 'ala
di pipistrello', 'bava di lumacone', 'forfora di serpente',
'rutto di orso', 'mandragora', 'cicuta', 'gas di palude']
```

Il mago è pronto per compiere qualche magia importante!

ELIMINARE ELEMENTI DA UNA LISTA

Per eliminare elementi da una lista, si usa il comando `del` (abbreviazione di delete, che significa "cancella"). Per esempio, per eliminare il sesto elemento nella lista del mago, la forfora di serpente, si può fare in questo modo:

```
>>> del mago_lista[5]
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'lingua di chiocciola',
'ala di pipistrello', 'bava di lumacone', 'rutto di orso',
'mandragora', 'cicuta', 'gas di palude']
```

NOTA

Ricordate che nelle liste le posizioni partono dall'indice zero, perciò `mago_lista[5]` *indica in effetti il sesto elemento della lista.*

Ed ecco come eliminare gli elementi che abbiamo appena aggiunto (mandragora, cicuta e gas di palude):

```
>>> del mago_lista[8]
>>> del mago_lista[7]
>>> del mago_lista[6]
>>> print(mago_lista)
['zampe di ragno', 'dito di rana', 'lingua di chiocciola', 'ala
di pipistrello', 'bava di lumacone', 'rutto di orso']
```

ARITMETICA DELLE LISTE

Possiamo unire (il termine più preciso è concatenare) due o più liste semplicemente sommandole, come faremmo con i numeri, mediante un segno più (`+`). Per esempio, supponiamo di avere due liste: `lista1`, che contiene i numeri da 1 a 4, e `lista2`, che contiene alcune parole. Possiamo unirle usando `print` e il segno `+`, in questo modo:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista2 = ['Sono', 'inciampato', 'e', 'ho', 'pestato', 'il',
'naso']
>>> print(lista1 + lista2)
[1, 2, 3, 4, 'Sono', 'inciampato', 'e', 'ho', 'pestato', 'il',
'naso']
```

È possibile anche unire (concatenare) due liste e assegnare il risultato a un'altra variabile:

```
>>> lista1 = [1, 2, 3, 4]
>>> lista2 = ['Ho', 'ancora', 'voglia', 'di', 'cioccolato']
>>> lista3 = lista1 + lista2
>>> print(lista3)
[1, 2, 3, 4, 'Ho', 'ancora', 'voglia', 'di', 'cioccolato']
```

Possiamo anche moltiplicare una lista per un numero. Per esempio, per moltiplicare `lista1` per 5, scriviamo `lista1 * 5`:

```
>>> lista1 = [1, 2]
>>> print(lista1 * 5)
[1, 2, 1, 2, 1, 2, 1, 2, 1, 2]
```

Questo equivale in effetti a dire a Python di ripetere `lista1` per cinque volte.

Non si può invece dividere una lista per un numero o sottrarre un numero da una lista: si ottengono solo messaggi di errore, come in questi esempi:

```
>>> lista1 / 20
Traceback (most recent call last):
  File "<pyshell#74>", line 1, in <module>
    lista1 / 20
TypeError: unsupported operand type(s) for /: 'list' and 'int'
>>> lista1 - 20
Traceback (most recent call last):
  File "<pyshell#75>", line 1, in <module>
    lista1 - 20
TypeError: unsupported operand type(s) for -: 'list' and 'int'
```

Perché? Beh, concatenare due liste con un + e ripetere una lista con un * sono operazioni abbastanza immediate. Questi concetti hanno un senso anche nel mondo reale. Per esempio, se vi porgessi due foglietti con due liste della spesa e vi dicessi, “Sommate queste due liste”, potreste scrivere le due liste su un altro foglio di carta in ordine, prima l’una e poi l’altra. Allo stesso modo, se vi dicessi “Moltiplicate questa lista per 3”, potreste immaginarvi di scrivere tutti gli elementi della lista per tre volte di fila su un altro foglietto.

Ma come si potrebbe dividere una lista? Per esempio, pensate a come potreste dividere una lista di sei numeri (da 1 a 6) in due. Questi sono solo tre modi diversi:

```
[1, 2, 3]     [4, 5, 6]
[1]    [2, 3, 4, 5, 6]
[1, 2, 3, 4]  [5, 6]
```

Dividereste la lista a metà, dopo il primo elemento o scegliereste un punto a caso in cui dividerla? Non esiste una risposta semplice e, se gli chiedete di dividere una lista, anche Python non sa che cosa fare. Per questo risponde con un messaggio di errore.

Lo stesso vale se si tenta di sommare a una lista qualche cosa che non sia un’altra lista. Non si può fare neanche questo. Per esempio, ecco che cosa succede se tentiamo di sommare il numero 50 alla `lista1`:

```
>>> lista1 + 50
Traceback (most recent call last):
  File "<pyshell#77>", line 1, in <module>
    lista1 + 50
TypeError: can only concatenate list (not "int") to list
```

Perché otteniamo un errore, in questo caso? Beh, che cosa vuol dire sommare 50 a una lista? Significa sommare 50 a ciascun elemento? Ma se gli elementi non sono numeri? Significa concatenare il numero 50 alla fine della lista o inserirlo

all’inizio?

Nella programmazione, i comandi devono funzionare esattamente nello stesso modo ogni volta che li si inserisce. Nella sua “stupidità”, il computer vede le cose solo in bianco e nero. Chiedetegli di prendere una decisione complicata, e lui alza le mani e dichiara un errore.

TUPLE

Una *tupla* è come una lista che usa le parentesi tonde, come in questo esempio:

```
>>> fib = (0, 1, 1, 2, 3)
>>> print(fib[3])
2
```

Qui abbiamo definito la variabile `fib` come contenente i numeri 0, 1, 1, 2 e 3. Poi, come nel caso di una lista, possiamo stampare l’elemento nella posizione di indice 3 nella tupla mediante `print(fib[3])`.

La differenza principale fra una tupla e una lista è che una tupla non può essere modificata, una volta che è stata creata. Per esempio, se tentiamo di sostituire il primo valore nella tupla `fib` con il numero 4 (come abbiamo sostituito valori nella nostra `mago_lista`), otteniamo un messaggio di errore:

```
>>> fib[0] = 4
Traceback (most recent call last):
  File “<pyshell#81>”, line 1, in <module>
    fib[0] = 4
TypeError: 'tuple' object does not support item assignment
```

Perché mai allora usare una tupla invece di una lista? Fondamentalmente perché a volte torna comodo usare qualcosa che si sa che non cambierà mai. Se si crea una tupla con due elementi, conterrà sempre quei due elementi.

LE MAPPE DI PYTHON NON VI AIUTERANNO A TROVARE LA STRADA

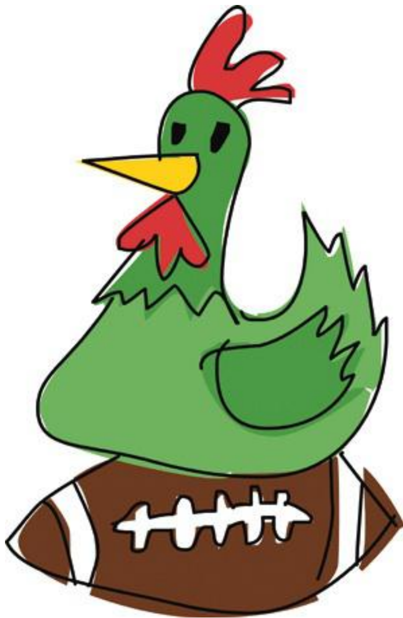
In Python, una *mappa* (qualche volta viene chiamata anche *dict*, abbreviazione di *dictionary*, ovvero dizionario) è un insieme di cose, come le liste e le tuple; la differenza è che in una mappa ogni elemento ha una chiave (*key*) e un corrispondente *valore*.

Per esempio, supponiamo di avere un elenco di persone con il loro sport preferito. Potremmo inserire tutte queste informazioni in una lista di Python, con il nome della persona seguito dallo sport preferito, per esempio in questo modo:

```
>>> sport_preferito = ['Ralph Williams, Football',
                      'Michael Tippett, Basketball',
                      'Edward Elgar, Baseball',
                      'Rebecca Clarke, Netball',
                      'Ethel Smyth, Badminton',
                      'Frank Bridge, Rugby']
```

Se vi chiedessi qual è lo sport preferito di Rebecca Clarke, potreste scorrere la lista e trovare la risposta: netball. Ma se la lista comprendesse cento persone (o molte di più)?

Se invece inseriamo le stesse informazioni in una mappa, con il nome della persona come chiave e il suo sport preferito come valore, il codice Python sarebbe come questo:



```
>>> sport_preferito = {'Ralph Williams' : 'Football',
                      'Michael Tippett' : ' Basketball',
                      'Edward Elgar' : 'Baseball',
                      'Rebecca Clarke' : 'Netball',
                      'Ethel Smyth' : 'Badminton',
                      'Frank Bridge' : 'Rugby'}
```

Usiamo il segno di due punti per separare ciascuna chiave dal suo valore; ciascuna chiave e ciascun valore sono racchiusi fra apici singoli. Notate, inoltre, che gli elementi di una mappa sono racchiusi fra parentesi graffe (`{}`), non tonde o quadre.

Il risultato è una mappa (ciascuna chiave viene "mappata", ovvero fatta corrispondere a un particolare valore), come indicato nella Tabella 3.1.

Tabella 3.1. Le chiavi puntano ai valori, in una mappa di sport preferiti.

Chiave	Valore
Ralph Williams	Football
Michael Tippett	Basketball
Edward Elgar	Baseball
Rebecca Clarke	Netball
Ethel Smyth	Badminton
Frank Bridge	Rugby

Ora, per scoprire quale sia lo sport preferito di Rebecca Clarke, possiamo accedere alla nostra mappa `sport_preferito` utilizzando il suo nome come chiave, in questo modo:

```
>>> print(sport_preferito['Rebecca Clarke'])
Netball
```

E la risposta è “netball”.

Per cancellare un valore in una mappa, si usa la chiave corrispondente. Per esempio, per eliminare Ethel Smyth:

```
>>> del sport_preferito['Ethel Smyth']
>>> print(sport_preferito)
{'Rebecca Clarke': 'Netball', 'Ralph Williams': 'Football',
'Frank Bridge': 'Rugby', 'Michael Tippett': ' Basketball',
'Edward Elgar': 'Baseball'}
```

Anche per sostituire un valore in una mappa si usa la chiave corrispondente:

```
>>> sport_preferito['Ralph Williams'] = 'Hockey'
>>> print(sport_preferito)
{'Rebecca Clarke': 'Netball', 'Ralph Williams': 'Hockey', 'Frank
Bridge': 'Rugby', 'Michael Tippett': ' Basketball', 'Edward
Elgar': 'Baseball'}
```

Abbiamo sostituito `Hockey` a `Football` come sport preferito di `Ralph Williams` usando il suo nome come chiave.

Lavorare con le mappe è un po’ come lavorare con liste e tuple, ma non si possono unire le mappe con l’operatore somma (`+`). Se ci provate, riceverete un messaggio di errore:

```
>>> sport_preferito = {'Ralph Williams' : 'Football',
                'Michael Tippett' : 'Basketball',
                'Edward Elgar' : 'Baseball',
                'Rebecca Clarke' : 'Netball',
                'Ethel Smyth' : 'Badminton',
                'Frank Bridge' : 'Rugby'}
>>> colore_preferito = {'Malcolm Warner' : 'Pallini rosa',
                 'James Baxter' : 'Strisce arancione',
                 'Sue Lee' : 'Viola pastello'}
>>> sport_preferito + colore_preferito
Traceback (most recent call last):
  File "<pyshell#101>", line 1, in <module>
    sport_preferito + colore_preferito
TypeError: unsupported operand type(s) for +: 'dict' and 'dict'
```

Il messaggio di errore dice che l'operatore + non può avere come operandi due mappe (*dict*). L'unione di due mappe non ha senso per Python, perciò come al solito alza le mani.

CHE COSA AVETE IMPARATO

In questo capitolo, avete imparato come Python usa le stringhe per conservare del testo e come usa liste e tuple per gestire più elementi. Avete visto che gli elementi nelle liste si possono modificare, e che si può accodare una lista a un'altra, mentre i valori in una tupla non si possono modificare.
Avete imparato anche come usare le mappe per conservare valori identificati da chiavi.

ROMPICAPO DI PROGRAMMAZIONE

Quelli che seguono sono alcuni esperimenti che potete provare a fare da voi. Potete trovare le risposte all'indirizzo *http://python-for-kids.com/*.

1. PREFERITI

Fate una lista degli hobby che preferite e date alla lista il nome di variabile `games`. Ora fate una lista dei vostri cibi preferiti e chiamate `foods` la variabile. Accodate le due liste e date al risultato il nome `favorites`. Infine, stampate la variabile `favorites`.

2. CONTARE I COMBATTENTI

Ci sono tre edifici con 25 ninja nascosti su ciascun tetto e due tunnel con 40 samurai nascosti in ciascuno. Quanti ninja e quanti samurai stanno per affrontarsi in battaglia? (Potete fare il calcolo con una formula nella shell di Python.)

3. SALUTI!

Create due variabili: una che punti al vostro cognome e una che punti al vostro nome. Ora create una stringa e usate i segnaposto per stampare il vostro nome con

un messaggio, utilizzando quelle due variabili, per esempio “Buona giornata, Brando Ickett!”.

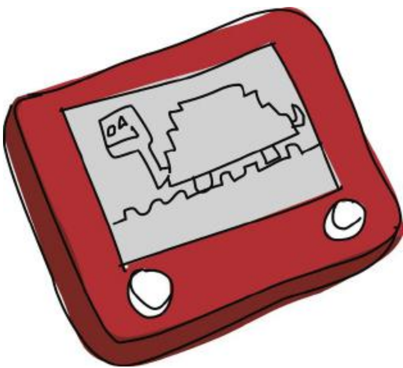
4
DISEGNARE CON LE TARTARUGHE

In Python, una tartaruga (*turtle*, in inglese) è un po' come una tartaruga nel mondo reale. Sappiamo che una tartaruga è un rettile che si sposta molto lentamente e porta la sua "casa" sul dorso. Nel mondo di Python, una tartaruga è una freccina nera che si sposta sullo schermo. In realtà, visto che una tartaruga di Python lascia una traccia spostandosi sullo schermo, assomiglia più a una chiocciola o a una lumaca.

La tartaruga è un modo elegante per imparare gli elementi fondamentali della grafica al calcolatore, perciò in questo capitolo useremo una tartaruga di Python per disegnare linee e figure semplici.

USARE IL MODULO TURTLE DI PYTHON

Un *modulo* in Python è un modo per mettere a disposizione un blocco di codice utile, che potrà essere utilizzato da un altro programma (fra le altre cose, il modulo può contenere funzioni che si possono impiegare liberamente). Parleremo meglio dei moduli nel Capitolo 7, ma intanto Python ha un modulo speciale, denominato turtle, che possiamo utilizzare per imparare come i computer disegnano immagini su uno schermo. Il modulo turtle permette di programmare grafica vettoriale, che fondamentalmente significa solo disegnare con linee, punti e curve semplici.



Vediamo come funziona turtle. Innanzitutto, avviate la shell di Python facendo clic sull'icona sul desktop (oppure, se usate Ubuntu, selezionate **IDLE** nel Software Center). Poi, dite a Python di usare la tartaruga importando il modulo `turtle`, in questo modo:

```
>>> import turtle
```

L'importazione di un modulo dice a Python che volete usarlo.

NOTA

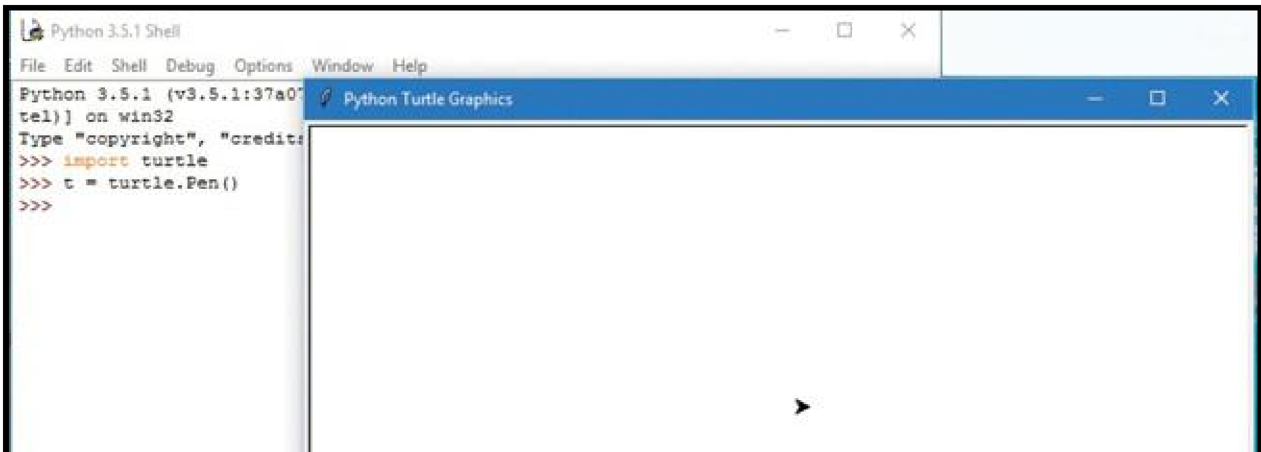
Se usate Ubuntu e a questo punto vi si presenta un messaggio di errore, dovete probabilmente installare `tkinter`. *Aprite il Software Center di Ubuntu e scrivete* python-tk *nella casella di ricerca. Nella finestra dovrebbe apparire la scritta "Tkinter – Writing Tk Applications with Python". Fate clic su Install per installare questo pacchetto.*

CREARE UN CANVAS

Ora che abbiamo importato il modulo `turtle`, dobbiamo creare un canvas (è il termine inglese per indicare una tela, cioè uno spazio su cui disegnare, come la tela di un pittore). Per crearlo, chiamiamo la funzione Pen del modulo turtle, che crea automaticamente un canvas (vedremo meglio più avanti che cosa sia una funzione). Inserite nella shell di Python quanto segue:

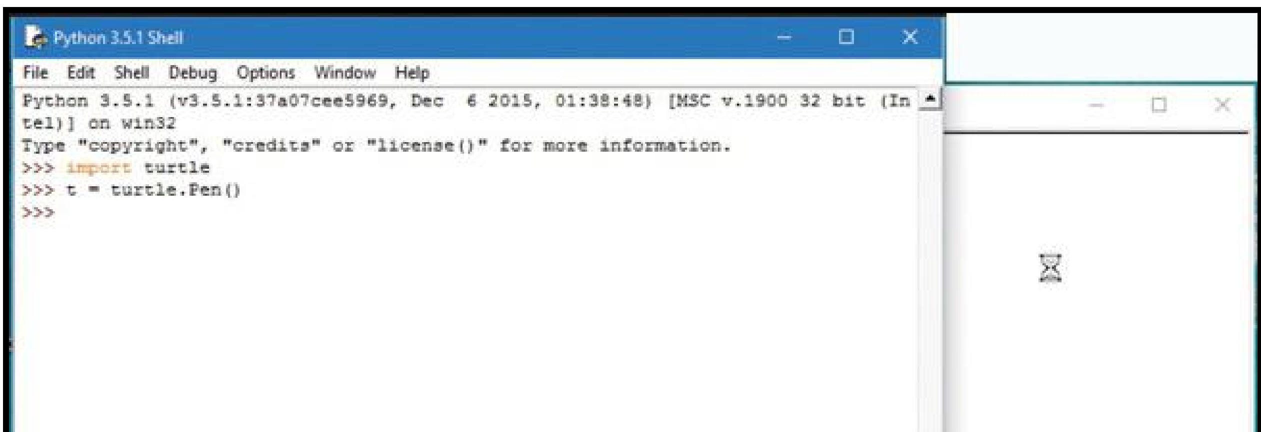
```
>>> t = turtle.Pen()
```

Dovrebbe comparire una finestra vuota (il canvas) con una freccia al centro, simile a questa:



La freccia al centro dello schermo è la tartaruga e sì, avete ragione, non assomiglia molto a una tartaruga.

Se la finestra *Turtle Graphics* si presenta dietro quella della Shell di Python, potrebbe darsi che non sembri funzionare correttamente. Se spostate il mouse sulla finestra Turtle Graphics, il cursore si trasforma in una clessidra:



Questo può succedere per varie ragioni: non avete avviato la shell dall'icona sul desktop (se usate Windows o un Mac), avete fatto clic su *IDLE (Python GUI)* nel menu Start di Windows, oppure IDLE non è stato installato correttamente. Provate a uscire e riavviare la shell dall'icona sul desktop.

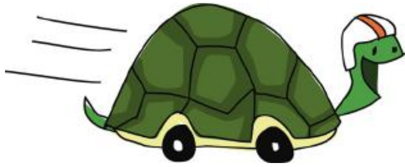
Se ancora non funziona tutto correttamente, provate a usare la console di Python invece della shell, in questo modo:

- In Windows, selezionate **Start** > **Tutti i programmi** (o Tutte le app) e poi, nel gruppo **Python 3.5**, fate clic su **Python 3.5**.
- In Mac OS X, fate clic sull'icona Spotlight nell'angolo superiore destro dello schermo e scrivete *Terminale* nella casella di input. Poi scrivete *python* quando si apre il terminale.

- In Ubuntu, aprite il terminale dal menu **Applications** e inserite *python*.

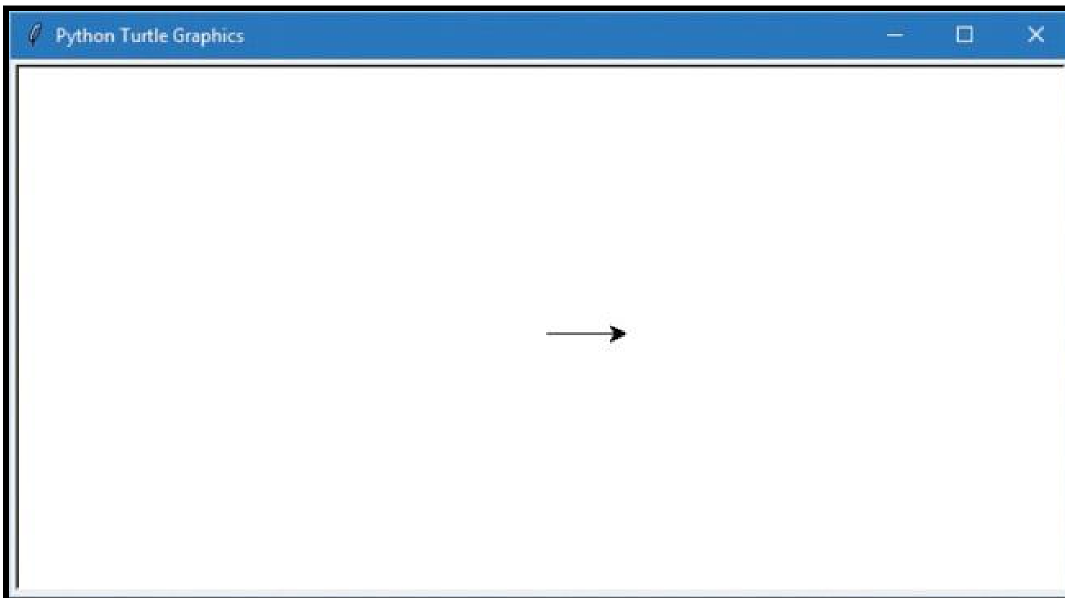
SPOSTARE LA TARTARUGA

Si inviano istruzioni alla tartaruga utilizzando le funzioni disponibili, applicate alla variabile `t` appena creata, come abbiamo usato la funzione `Pen` nel modulo `turtle`. Per esempio, l'istruzione `forward` dice alla tartaruga di spostarsi in avanti. Per dire alla tartaruga di avanzare di 50 pixel, inserite questo comando:

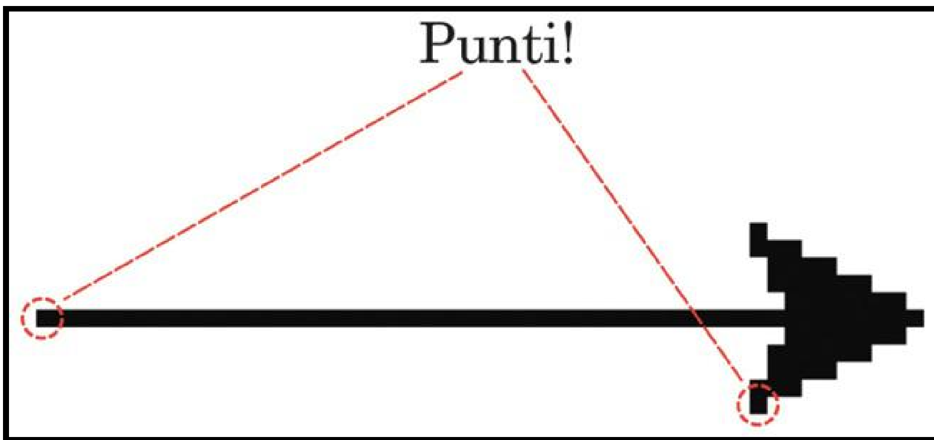


```
>>> t.forward(50)
```

Dovreste vedere qualcosa di simile a questo:



La tartaruga si è spostata in avanti di 50 pixel. Un pixel è un singolo punto sullo schermo, l'elemento più piccolo che può essere rappresentato. Tutto quello che vedete sul monitor del vostro computer è costituito da pixel, che sono piccoli punti quadrati. Se poteste ingrandire il canvas e il segmento disegnato dalla tartaruga, potreste vedere che la freccia che rappresenta il percorso della tartaruga è solo una serie di pixel. Questa è semplice grafica al computer.

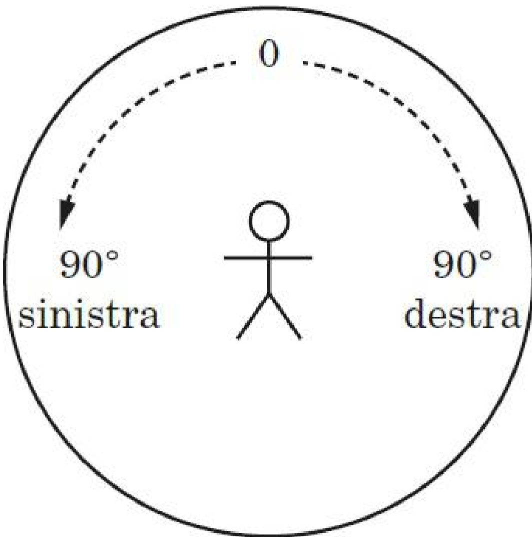


Ora dite alla tartaruga di ruotare a sinistra di 90 gradi con questo comando:

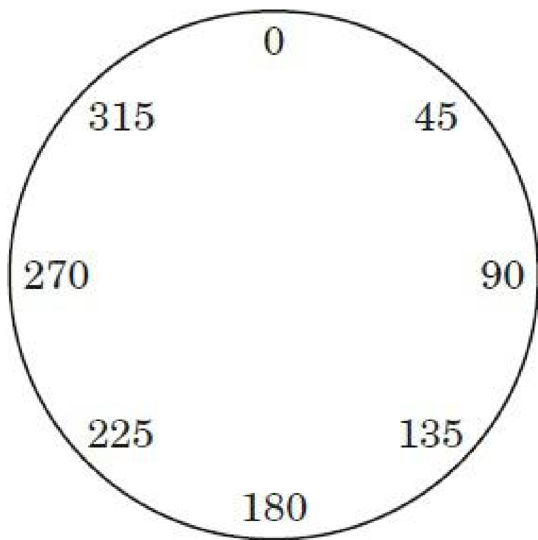
```
>>> t.left(90)
```

Se non avete ancora studiato i gradi, immaginatevi di essere in piedi al centro di un cerchio.

- La direzione in cui siete rivolti è 0 gradi.
- Se estendete il braccio sinistro, quelli sono 90 gradi (90°) a sinistra.
- Se estendete il braccio destro, quelli sono 90 gradi (90°) a destra.



Continuando a procedere lungo il cerchio a destra, dal punto indicato dal vostro braccio destro, 180 gradi sono direttamente alle vostre spalle, 270 gradi sono la direzione in cui punta il vostro braccio sinistro e 360 gradi vi riportano al punto di partenza: i gradi vanno da 0 a 360. Ecco qui sotto i gradi in un cerchio completo, ruotando verso destra (in senso orario), riportati per incrementi di 45 gradi.



Quando la tartaruga di Python si gira a sinistra, si dispone in modo da puntare verso la nuova direzione (come avete ruotato il vostro corpo per vedere dove puntava il vostro braccio esteso, 90 gradi a sinistra).

Il comando `t.left(90)` fa sì che la freccia punti verso l'alto (dato che inizialmente puntava verso destra).

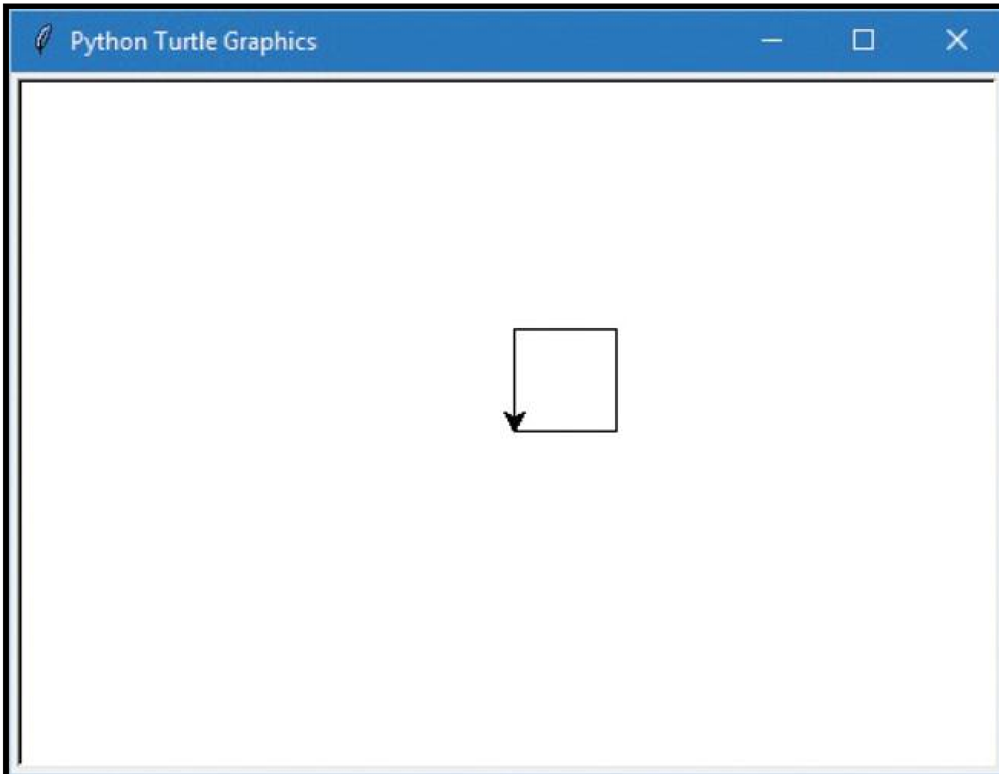
NOTA

Il comando `t.left(90)` *equivale a* `t.right(270)`. *Analogamente,* `t.right(90)` *equivale a* `t.left(270)`. *Immaginatevi il cerchio e seguite la sua circonferenza con i gradi.*

Ora disegneremo un quadrato. Aggiungete questo codice alle righe che avete già inserito:

```
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
>>> t.forward(50)
>>> t.left(90)
```

La vostra tartaruga ha disegnato un quadrato e ora sarà rivolta nella stessa direzione dell'inizio.



Per cancellare il canvas, scrivete `reset`. Questo pulisce la tela e riporta la tartaruga nella sua posizione iniziale.

```
>>> t.reset()
```

Potete usare anche la funzione `clear`, che si limita a cancellare tutto nella finestra e lascia la tartaruga nello stesso punto in cui si trova.

```
>>> t.clear()
```

Si può far girare la tartaruga anche a destra (`right`) o farla muovere a ritroso (`backward`). Si può usare `up` (su) per sollevare la penna dalla pagina (in altre parole, per dire alla tartaruga di smettere di disegnare) e `down` (giù) per iniziare a disegnare. Queste funzioni sono scritte nello stesso modo delle altre che abbiamo già usato.

Proviamo a fare un altro disegno utilizzando alcuni di questi comandi. Questa volta, faremo disegnare alla tartaruga due segmenti.

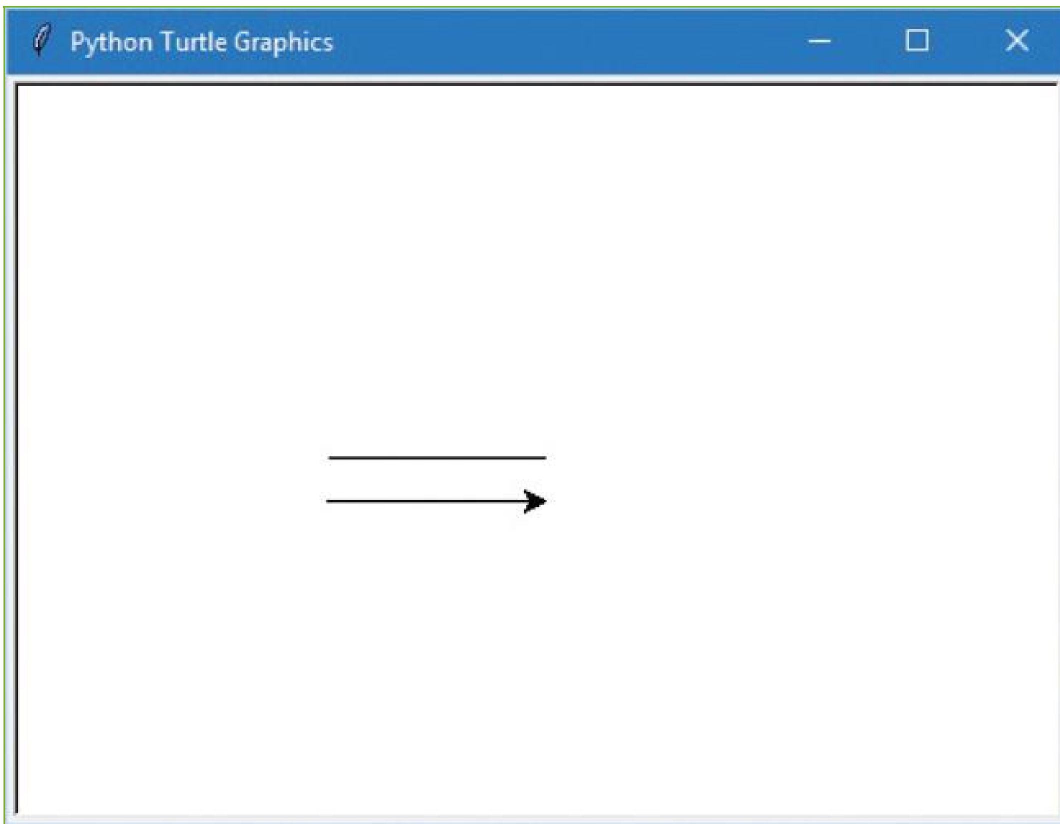
Inserite questo codice:

```
>>> t.reset()
>>> t.backward(100)
>>> t.up()
>>> t.right(90)
>>> t.forward(20)
>>> t.left(90)
>>> t.down()
>>> t.forward(100)
```

Come prima cosa abbiamo reinizializzato la tela e riportato la tartaruga al suo punto di partenza con `t.reset()`; poi abbiamo spostato la tartaruga all’indietro di 100 pixel con `t.backward(100)`, poi usato `t.up()` per sollevare la penna e smettere di disegnare.

Poi, con il comando `t.right(90)`, abbiamo fatto ruotare la tartaruga a destra di 90 gradi, in modo che si orientasse verso il basso sullo schermo, poi con `t.forward(20)` l’abbiamo fatta spostare in avanti di 20 pixel. Non è stato disegnato nulla in questo passo, perché avevamo dato il comando `up` nella terza riga.

Poi abbiamo fatto ruotare la tartaruga a sinistra di 90 gradi, con `t.left(90)` in modo che fosse orientata verso destra sullo schermo; abbiamo dato il comando `down` per dire alla tartaruga di riabbassare la penna e ricominciare a disegnare. Infine, abbiamo disegnato un segmento in avanti, parallelo al primo segmento disegnato, con `t.forward(100)`. I due segmenti paralleli che abbiamo disegnato si presenteranno così:



CHE COSA AVETE IMPARATO

In questo capitolo, avete imparato come usare il modulo `turtle` di Python. Abbiamo disegnato alcuni semplici segmenti di retta, utilizzando i comandi `left` (sinistra) e `right` (destra), `forward` (avanti) e `backward` (indietro). Avete visto come dire alla tartaruga di non disegnare, con il comando `up` (su) e di ricominciare a disegnare, con il comando `down` (giù). Avete anche scoperto che la tartaruga si gira per gradi.

ROMPICAPO DI PROGRAMMAZIONE

Provate a disegnare con la tartaruga qualcuna delle forme seguenti. Potete trovare le risposte all'indirizzo *http://python-for-kids.com/*.

1. UN RETTANGOLO

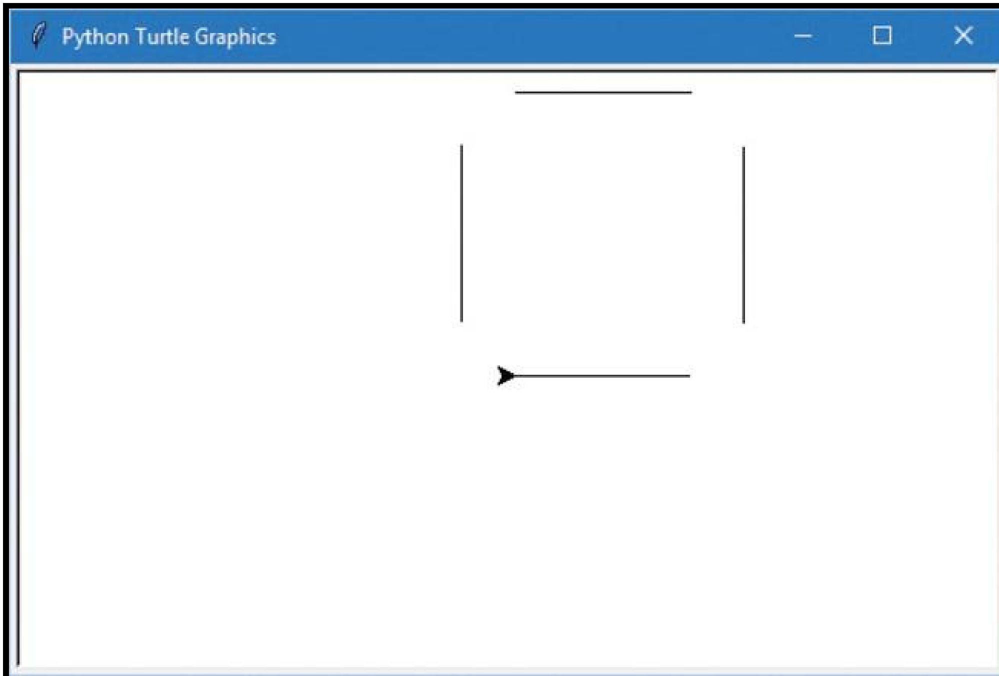
Create un nuovo canvas con la funzione `Pen` del modulo `turtle` e poi disegnate un rettangolo.

2. UN TRIANGOLO

Create un altro canvas e questa volta disegnate un triangolo. Riguardate il diagramma del cerchio su cui avevamo riportato i gradi (a pagina 48) per stabilire in quale direzione far ruotare la tartaruga utilizzando i gradi.

3. UN RIQUADRO SENZA ANGOLI

Scrivete un programma che disegni i quattro segmenti mostrati qui (non sono importanti le dimensioni, quello che conta è la forma):



5

PORRE DOMANDE CON IF E ELSE

Nella programmazione, si pongono spesso domande a cui si può rispondere con un sì o un no, e si decide di fare qualcosa a seconda della risposta. Per esempio, potremmo chiedervi: “Avete più di 20 anni?” e, se la risposta fosse sì, rispondervi con un “Siete troppo vecchi!”.

Questi tipi di domande si definiscono *condizioni* e condizioni e risposte si combinano in *enunciati* `if` (in inglese, *if* corrisponde al nostro se). Le condizioni possono essere più complesse di una singola domanda e gli enunciati `if` si possono combinare con più domande e differenti decisioni a seconda della risposta a ciascuna domanda.

In questo capitolo, vedremo come usare gli enunciati `if` nella costruzione di programmi.

ENUNCIATI IF

Un enunciato `if` può essere scritto in Python in questo modo:

```
>>> età = 13
>>> if età > 20:
        print('Sei troppo vecchio!')
```

Un enunciato `if` è costituito dalla parola chiave `if`, seguita da una condizione e da un segno di due punti (`:`), come in `if età > 20:`. Le righe che seguono i due punti devono essere in un blocco e, se la risposta alla domanda è sì (o *vero*, come si dice di solito quando si programma in Python), i comandi presenti nel blocco vengono eseguiti. Ora, esploriamo come scrivere blocchi e condizioni.

UN BLOCCO È UN GRUPPO DI ENUNCIATI DI PROGRAMMAZIONE

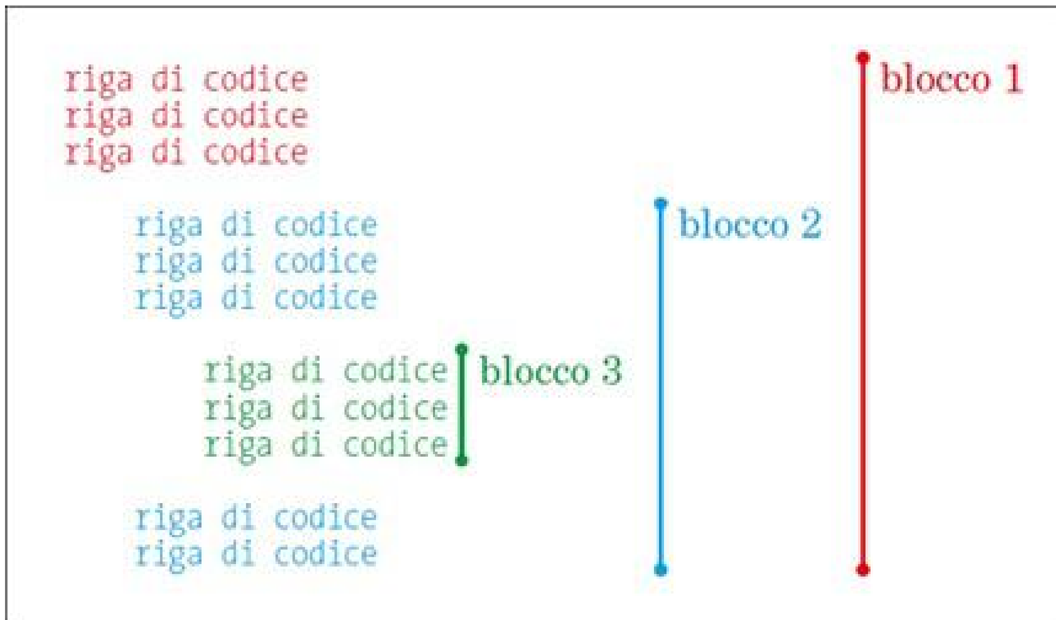
Un *blocco* di codice è un insieme di enunciati di programmazione raggruppati. Per esempio, se `if età > 20:` è vero, potreste volere fare qualcosa di più che stampare semplicemente la frase "Sei troppo vecchio!".
Potreste per esempio stampare qualche altra frase opportuna, come in questo caso:

```
>>> if età > 20:
        print('Sei troppo vecchio!')
        print ('Che cosa ci fai qui?')
        print ('Perché non sei a pulire il giardino?')
```

Questo blocco di codice è costituito da tre enunciati `print` che vengono eseguiti solo se la condizione `età > 20` è vera. Ciascuna riga nel blocco inizia con quattro spazi, se la confrontate con l'enunciato `if` precedente. Riguardiamo il codice, questa volta visualizzando gli spazi:

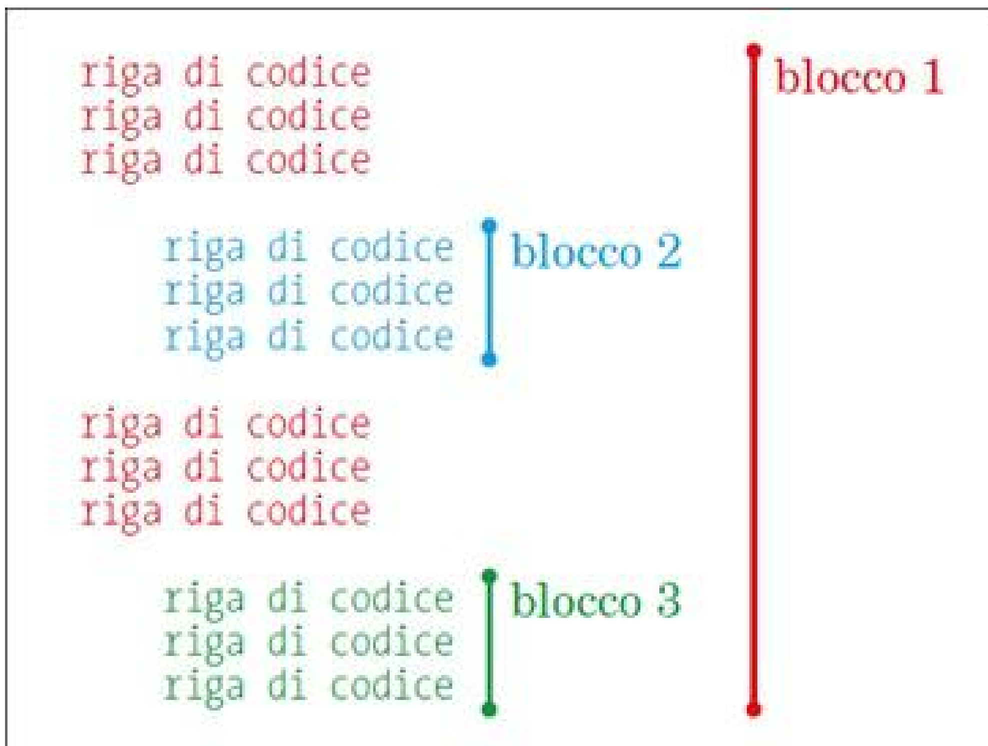
```
>>> if età > 20:
      □□□□print('Sei troppo vecchio!')
      □□□□print ('Che cosa ci fai qui?')
      □□□□print ('Perché non sei a pulire il giardino?')
```

In Python, lo spazio bianco (*whitespace*), per esempio una tabulazione (che si inserisce premendo il tasto TAB) o uno spazio (inserito quando si preme la barra spaziatrice), è dotato di significato. Il codice che si trova nella stessa posizione (rientrato dello stesso numero di spazi rispetto al margine sinistro) è raggruppato in un blocco e, ogni volta che si inizia una nuova riga con più spazi della precedente, si inizia un nuovo blocco che fa parte del precedente, in questo modo:



Si raggruppano enunciati in blocchi perché sono in relazione fra loro e devono essere eseguiti insieme.

Quando si cambia il rientro (in gergo si chiama anche indentazione), in genere si creano nuovi blocchi. L’esempio seguente mostra tre blocchi separati che vengono creati semplicemente modificando il rientro.



Qui, anche se i blocchi 2 e 3 hanno lo stesso rientro, sono considerati blocchi diversi perché in mezzo vi è un blocco con un rientro minore (meno spazi bianchi iniziali).

Un blocco con quattro spazi in una riga e sei nella successiva genererà un errore (*indentation error*), quando lo si esegue, perché Python si aspetta che venga usato lo stesso numero di spazi per tutte le righe di un blocco. Così, se iniziate un blocco con quattro spazi, dovete usare sempre quattro spazi per quel blocco. Ecco un esempio:

```
>>> età = 25
>>> if età > 20:
        □□□□print('Sei troppo vecchio!')
        □□□□□□print('Che cosa ci fai qui?')
```

Ho reso visibili gli spazi in modo che possiate vedere le differenze: notate che la terza riga ha sei spazi e non quattro.

Quando si prova a eseguire questo codice, IDLE evidenzia con un rettangolo rosso la riga in cui vede un problema e mostra un messaggio `SyntaxError` con la spiegazione (*unexpected indent*, ovvero “rientro inatteso”):

```
>>> età = 25
>>> if età > 20:
        print('Sei troppo vecchio!')
        █print('Che cosa ci fai qui?')
SyntaxError: unexpected indent
```

Python non si aspettava di vedere due spazi in più all’inizio della seconda riga `print`.

NOTA

L'uso di spazi coerenti rende più facile la lettura del codice. Se cominciate a scrivere un programma e inserite quattro spazi all'inizio di un blocco, continuate a usare quattro spazi all'inizio degli altri blocchi. Inoltre, fate attenzione a rientrare ogni riga di uno stesso blocco con lo stesso numero di spazi.

LE CONDIZIONI AIUTANO A FARE CONFRONTI

Una *condizione* è un enunciato che confronta cose e ci dice se ciò che è indicato nel confronto è `True` (vero, sì) oppure `False` (falso, no). Per esempio, `età > 10` è una condizione, ed è un modo per chiedere "Il valore della variabile `età` è maggiore di 10?". Anche questa è una condizione: `colore_capelli == 'rosso'`, ovvero: "Il valore della variabile `colore_capelli` è `rosso`?".

In Python usiamo dei simboli (chiamati *operatori*) per creare le condizioni, come "maggiore di", "uguale a", "minore di". La Tabella 5.1 elenca alcuni di questi simboli.

Tabella 5.1 Simboli per le condizioni.

Simbolo	Definizione
==	Uguale a
!=	Non uguale a (diverso da)
>	Maggiore di
<	Minore di
>=	Maggiore o uguale a
<=	Minore o uguale a

Per esempio, se avete 10 anni, la condizione `tua_età == 10` restituirebbe `True`; altrimenti restituirebbe `False`. Se avete 12 anni, la condizione `tua_età > 10` restituirebbe `True`.

ATTENTI

State attenti a usare un doppio segno di uguale (==) quando definite una condizione "uguale a".

Vediamo qualche altro esempio. Qui, poniamo la vostra età a 10 anni e poi scriviamo un enunciato condizionale che stamperà "Sei troppo vecchio per le mie battute!" se l'età è maggiore di 10.

```
>>> età = 10
>>> if età > 10:
        print('Sei troppo vecchio per le mie battute!')
```

Che cosa succede quando lo scriviamo in IDLE e poi premiamo INVIO?

Niente, perché il valore della variabile `età` non è maggiore di 10, quindi Python non esegue il blocco `print`. Se avessimo invece impostato la variabile `età` al valore 20, il messaggio sarebbe stato stampato.

Ora cambiamo l'esempio precedente utilizzando una condizione "maggiore o uguale a" (`>=`):

```
>>> età = 10
>>> if età >= 10:
        print('Sei troppo vecchio per le mie battute!')
```

Questa volta vedrete stampato "Sei troppo vecchio per le mie battute!" sullo schermo, perché il valore della variabile età è uguale a 10.

Ora proviamo utilizzando una condizione "uguale a" (`==`);

```
>>> età = 10
>>> if età == 10:
        print('Che cos\'è marrone e appiccicoso?')
```

Questa volta il messaggio comparirà sullo schermo.

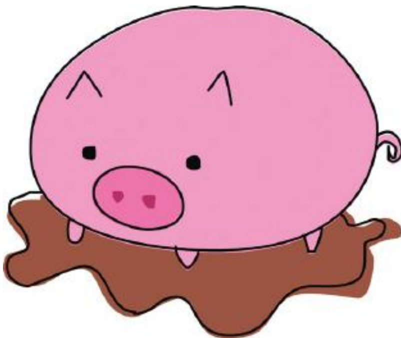
ENUNCIATI IF-THEN-ELSE

Oltre a usare enunciati `if` per fare qualcosa quando viene soddisfatta (è `True`) una condizione, possiamo usarli anche per fare qualcosa quando una condizione non è vera. Per esempio, potremmo stampare sullo schermo un messaggio se l'età è 12 (`True`) e un altro se non è 12 (`False`). Il trucco sta nell'usare un enunciato `if-then-else` (se-allora-altrimenti), che sostanzialmente dice: "Se (*if*) una certa cosa è vera, allora (*then*) fa questo; altrimenti (*else*) fai quest'altro".

Creiamo un enunciato `if-then-else`. Scrivete quanto segue nella shell:

```
>>> print('Vuoi sentire una barzelletta sporca?')
Vuoi sentire una barzelletta sporca?
>>> age = 12
>>> if age == 12:
        print('Un maialino è cascato nel fango!')
else:
        print('Shh. È un segreto!')

Un maialino è cascato nel fango!
```

Dato che abbiamo impostato la variabile età al valore 12 e la condizione chiede se l'età sia uguale a 12, dovreste vedere sullo schermo il messaggio del primo `print`. Ora provate a cambiare il valore dell'età in un numero diverso da 12, per esempio:



```
>>> print('Vuoi sentire una barzelletta sporca?')
Vuoi sentire una barzelletta sporca?
>>> age = 8
>>> if age == 12:
        print('Un maialino è cascato nel fango!')
else:
        print('Shh. È un segreto!')

Shh. È un segreto.
```

Questa volta vedrete stampato il secondo messaggio.

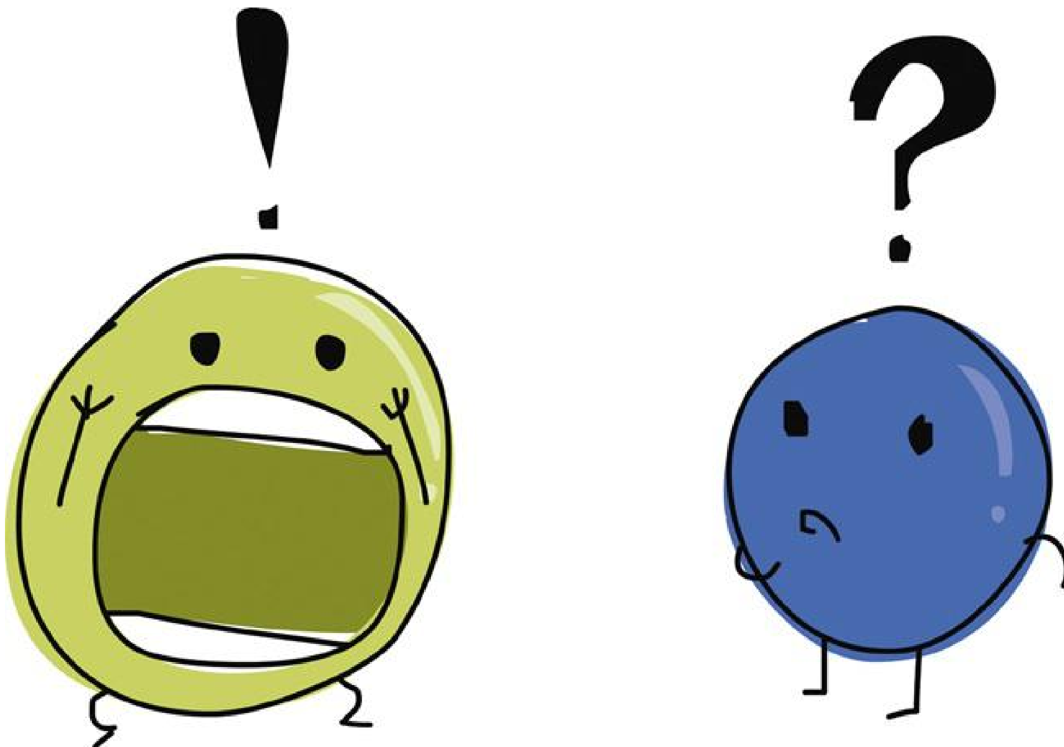
ENUNCIATI IF ED ELIF

Possiamo estendere ulteriormente un enunciato `if` con `elif` (che è una abbreviazione di `else-if`, "altrimenti-se"). Per esempio, possiamo controllare se l'età di una persona è 10, 11 o 12 anni (e così via) e fare in modo che il programma faccia qualcosa di diverso a seconda della risposta. Questi enunciati sono diversi dagli `if-then-else`, perché possono esserci più `elif` nello stesso enunciato:

```
   >>> età = 12
❶ >>> if età == 10:
❷         print ('Che cosa fa un orso nella steppa?')
          print ('Cammina!')
❸ elif età == 11:
          print('Che cosa dice l\'uva bianca all\'uva blu?')
          print('Respira! Respira!')
❹ elif età == 12:
❺         print('Che cosa ha detto lo 0 a 8?')
          print('Ciao, ragazzi!')
  elif età == 13:
          print('Perché 10 non ha paura di 7?')
          print('Perché 9 qui non piove!')
  else:
          print('Beh?')

  Che cosa ha detto lo 0 a 8? Ciao, ragazzi!
```

In questo esempio, l'enunciato `if` nella seconda riga controlla se il valore della variabile `età` è uguale a 10 in ❶. L'enunciato `print` che segue in ❷ viene eseguito se `età` è uguale a 10. Però, poiché abbiamo attribuito a `età` il valore 12, il computer salta al successivo enunciato `if` in ❸ e controlla se il valore di `età` è uguale a 11. Non lo è, perciò salta al successivo enunciato `if` in ❹ e vede se `età` è uguale a 12. Lo è, perciò questa volta esegue il comando `print` in ❺.

Quando inserite questo codice in IDLE, viene automaticamente rientrato, perciò ricordate di premere il tasto CANC o il BACKSPACE dopo aver scritto gli enunciati `print`, in modo che gli `if`, `elif` ed `else` inizino dal margine sinistro. Questa è la stessa posizione in cui sarebbe l'enunciato `if` se non ci fosse il prompt (`>>>`).



COMBINARE LE CONDIZIONI

Le condizioni si possono combinare utilizzando le parole chiave `and` e `or` (che

corrispondono rispettivamente alla congiunzione “e” e alla disgiunzione “o”): così il codice può essere più breve e più semplice. Ecco un esempio con `or`:

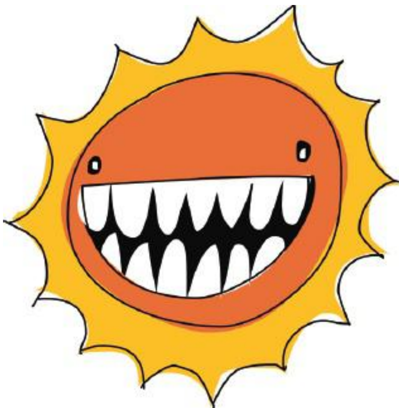
```
>>> if età == 10 or età == 11 or età == 12 or età == 13:
        print('Cosa fa 13 + 49 + 84 + 155 + 97? Un mal di testa!')
else:
        print('Cosa?')
```

In questo codice, se una qualsiasi delle condizioni nella prima riga è vera (in altre parole, se la variabile `età` ha valore 10, 11, 12 *o* 13), viene eseguito il blocco di codice che inizia con `print` nella riga successiva. Se nessuna delle condizioni nella prima riga è vera (`else`), Python passa al blocco dell’ultima riga, e visualizza sullo schermo `Cosa?`.

Per compattare di più, potremmo usare `and` e gli operatori “maggiore o uguale a” (`>=`) e “minore o uguale a” (`<=`):

```
>>> if età >=10 and età <=13:
        print('Cosa fa 13 + 49 + 84 + 155 + 97? Un mal di testa!')
else:
        print('Cosa?')
```

Qui, se `età` è maggiore o uguale a 10 e (*and*) minore o uguale a 13 (condizione definita nella prima riga con l’enunciato `if età >= 10 and età <= 13:`), viene eseguito il blocco di codice che inizia con `print` sulla riga successiva. Per esempio, se il valore di `età` è 12, verrà stampato sullo schermo `Cosa fa 13 + 49 + 84 + 155 + 97? Un mal di testa!`, perché 12 è maggiore di 10 e minore di 13.



VARIABILI SENZA VALORE: NONE

A una variabile, come abbiamo visto, si possono assegnare valori, stringhe e liste, ma si può anche assegnare un valore vuoto, nullo. In Python un valore vuoto è indicato come `None` (“niente”), e indica l’assenza di valore. È importante notare che il valore `None` è diverso dal valore 0 perché indica l’assenza di un valore, non un numero con valore 0. L’unico valore che una variabile ha quando le si assegna il valore vuoto `None` è… niente. Ecco un esempio:

```
>>> mioval = None
>>> print(mioval)
None
```

Assegnare un valore `None` a una variabile è un modo per dire che quella variabile non contiene più alcun valore (o meglio, che non è più l'etichetta di un valore). Impostare una variabile a `None` è anche un modo per definire una variabile senza indicarne il valore. Lo si può fare quando si sa che si avrà bisogno di una variabile in seguito nel corso del programma, ma si vuole definire inizialmente tutte le variabili. I programmatori definiscono spesso le loro variabili all'inizio di un programma perché, collocate in quella posizione, è facile vedere i nomi di tutte le variabili utilizzate in un certo frammento di codice.

Si può anche controllare se una variabile non ha valore in un enunciato `if`, come in questo esempio:

```
>>> mioval = None
>>> if mioval == None:
        print('La variabile mioval non ha alcun valore')

La variabile mioval non ha alcun valore
```

Un'istruzione del genere è utile quando si vuole calcolare un valore per una variabile solo se non è stato già calcolato.

LA DIFFERENZA FRA STRINGHE E NUMERI

Input dell'utente è quello che una persona scrive alla tastiera – non importa se si tratta di un carattere, della pressione di un tasto freccia o di INVIO o di qualsiasi altra cosa.

L'input dell'utente viene recepito da Python come una stringa, il che significa che, se scrivete sulla tastiera il numero 10, Python salva il 10 in una variabile come stringa e non come numero.

Qual è la differenza fra il numero 10 e la stringa `'10'`? A noi sembrano la stessa cosa, con l'unica differenza che la seconda è racchiusa fra apici. Per un computer, però, si tratta di due cose molto diverse.

Per esempio, supponiamo di confrontare il valore della variabile `età` con un numero in un enunciato `if`:

```
>>> if età == 10:
        print("Qual è il modo migliore per parlare a un mostro?")
        print("Da più lontano possibile!")
```

Poi impostiamo la variabile `età` al numero `10`:

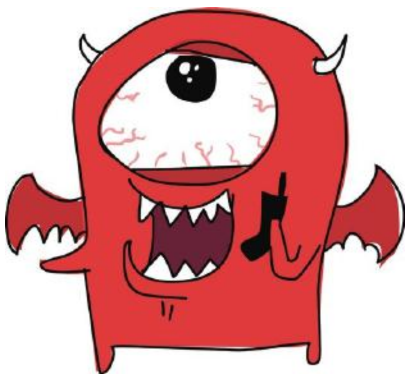
```
>>> età = 10
>>> if età == 10:
        print("Qual è il modo migliore per parlare a un mostro?")
        print("Da più lontano possibile!")

Qual è il modo migliore per parlare a un mostro?
Da più lontano possibile!
```

Come potete vedere, l’enunciato `print` viene eseguito.

Ora impostiamo la variabile `età` alla stringa `'10'` (con gli apici), in questo modo:

```
>>> età = '10'
>>> if età == 10:
        print("Qual è il modo migliore per parlare a un mostro?")
        print("Da più lontano possibile!")
```

Ora il codice nell’enunciato `print` non viene eseguito, perché Python non vede il numero fra apici (una stringa) come un numero. Per fortuna, Python possiede alcune funzioni magiche che trasformano stringhe in numeri e numeri in stringhe. Per esempio, si può convertire la stringa `'10'` in un numero con la funzione `int`:



```
>>> età = '10'
>>> età_convertita = int(età)
```

La variabile `età_convertita` ora contiene il numero 10.

Per convertire un numero in una stringa si usa invece la funzione `str`:

```
>>> età = 10
>>> età_convertita = str(età)
```

In questo caso, `età_convertita` conterrà la stringa `'10'` anziché il numero 10.

Ricordate che l’enunciato `if età == 10` non ha stampato nulla quando la variabile aveva come valore una stringa `(età = '10')`? Se però prima convertiamo la variabile, otteniamo un risultato del tutto diverso:

```
>>> età = '10'
>>> età_convertita = int(età)
>>> if età_convertita == 10:
        print("Qual è il modo migliore per parlare a un mostro?")
        print("Da più lontano possibile!")

Qual è il modo migliore per parlare a un mostro?
Da più lontano possibile!
```

Ma fate attenzione: se tentate di convertire un numero decimale, otterrete un errore, perché la funzione `int` si aspetta un intero:

```
>>> età = '10.5'
>>> età_convertita = int(età)
Traceback (most recent call last):
  File "<pyshell#37>", line 1, in <module>
    età_convertita = int(età)
ValueError: invalid literal for int() with base 10: '10.5'
```

`ValueError` è il modo che usa Python per dirvi che il valore che avete tentato di utilizzare non è appropriato. In casi del genere dovete usare la funzione `float` al posto di `int`. La funzione `float` può trattare numeri che non sono interi.

```
>>> età = '10.5'
>>> età_convertita = float(età)
>>> print(età_convertita)
10.5
```

Otterrete un `ValueError` anche se tentate di convertire in cifre una stringa che non contiene un numero:

```
>>> età = 'dieci'
>>> età_convertita = int(età)
Traceback (most recent call last):
  File "<pyshell#45>", line 1, in <module>
    età_convertita = int(età)
ValueError: invalid literal for int() with base 10: 'dieci'
```

CHE COSA AVETE IMPARATO

In questo capitolo, avete imparato come lavorare con gli enunciati `if` per creare blocchi di codice che vengono eseguiti solo quando determinate condizioni sono vere. Avete visto come estendere gli enunciati `if` con `elif` in modo da eseguire sezioni diverse di codice a seconda del risultato di condizioni diverse, e come usare la parola chiave `else` per eseguire del codice quando nessuna delle condizioni risulta vera. Avete imparato anche a combinare condizioni utilizzando le parole chiave `and` e `or`, in modo da vedere per esempio se un numero ricade entro un certo intervallo, e come trasformare stringhe in numeri (e viceversa) con `int`, `str` e `float`. E avete scoperto che il nulla (`None`) ha un significato in Python e può essere utilizzato per riportare le variabili al loro stato iniziale, vuoto.

ROMPICAPO DI PROGRAMMAZIONE

Provate a risolvere questi rompicapo utilizzando l'enunciato `if` e le condizioni. Potete trovare le risposte all'indirizzo *http://python-for-kids.com/*.

1. SEI RICCO?

Che cosa pensate che faccia, il codice seguente? Provate a dare una risposta senza scriverlo nella shell, poi controllate la vostra risposta.

```
>>> denaro = 2000
>>> if denaro > 1000:
        print('Sono ricco!')
else:
        print('Non sono ricco!!')
          print('Ma magari in futuro...')
```

2. TWINKIES!

Create un enunciato `if` che verifichi se il numero di Twinkies (nella variabile `twinkies`) è minore di 100 o maggiore di 500. Il programma dovrà stampare il messaggio "Troppo pochi o troppo tanti" se la condizione è vera. (Che cosa sono i Twinkies? Dategli voi un significato…)

3. IL NUMERO GIUSTO

Create un enunciato `if` che verifichi se la quantità contenuta nella variabile denaro sia compresa fra 100 e 500 *o* fra 1000 e 5000.

4. POSSO COMBATTERE CONTRO I NINJA

Create un enunciato `if` che stampi la stringa "Sono troppi" se la variabile `ninja` contiene un numero minore di 50, stampi "Sarà una bella lotta, ma posso farcela", se il valore è inferiore a 30 e stampi "Posso combattere contro quei ninja!" se è inferiore a 10. Potete provare il vostro codice con:

```
>>> ninja = 5
```

6
GIRARE IN TONDO

Non c'è niente di peggio che dover fare la stessa cosa più e più volte. C'è un motivo per cui qualcuno si mette a contare le pecore quando ha difficoltà ad addormentarsi e non ha nulla a che fare con supposte doti "dormitive" di questi ovini: è perché ripetere senza fine qualche cosa è noioso, e la nostra mente può cadere nel sonno più facilmente se non ci si concentra su qualcosa di interessante.

Neanche i programmatori amano particolarmente ripetersi, a meno che non stiano cercando di addormentarsi. Per fortuna, la maggior parte dei linguaggi di programmazione mette a disposizione quello che è chiamato ciclo `for` (*for* è l'equivalente dell'italiano "per"), che ripete altri enunciati e blocchi di codice automaticamente.

In questo capitolo, vedremo i cicli `for` e un altro tipo di ciclo che ci mette a disposizione Python: il ciclo `while` (parola che corrisponde all'italiano "mentre").

USARE I CICLI FOR

Per stampare cinque volte "ciao" in Python, si potrebbe scrivere:

```
>>> print('ciao')
ciao
>>> print('ciao')
ciao
>>> print('ciao')
ciao
>>> print('ciao')
ciao
>>> print('ciao')
ciao
```

Sarebbe un po' noioso. Si può usare un ciclo `for` per scrivere di meno e ridurre le ripetizioni, in questo modo:

```
❶ >>> for x in range(0, 5):
❷     print('ciao')

ciao
ciao
ciao
ciao
ciao
```

La funzione `range` (intervallo) in ❶ si usa per creare un elenco di numeri che vanno da quello iniziale fino a quello che precede il numero finale. Può sembrare un po' confuso. Combiniamo la funzione `range` con la funzione `list` per vedere esattamente come funziona. La funzione `range` non crea in realtà una lista di numeri, ma restituisce un *iteratore*, che è un tipo di oggetto Python progettato specificamente per lavorare con i cicli. Se però combiniamo `range` con `list`, otteniamo una lista di numeri.

```
>>> print(list(range(10, 20)))
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
```

Nel caso del ciclo `for`, il codice nella riga ❶ dice in sostanza a Python di fare quanto segue:

- Inizia a contare da 0 e fermati prima di raggiungere 5.
- Per ogni numero che conti, immagazzina il valore nella variabile `x`.

Poi Python esegue il blocco di codice in ❷.
Notate che ci sono quattro spazi aggiuntivi all'inizio della riga ❷ (rispetto alla riga ❶). IDLE ha automaticamente rientrato questo blocco.
Quando premiamo INVIO dopo la seconda riga, Python stampa "ciao" cinque volte.
Potremmo anche usare la `x` nell'enunciato `print` per contare i "ciao":

```
>>> for x in range(0, 5):
        print('ciao %s' % x)
ciao 0
ciao 1
ciao 2
ciao 3
ciao 4
```

Se non avessimo usato il ciclo `for`, il nostro codice avrebbe dovuto essere qualcosa di questo genere:

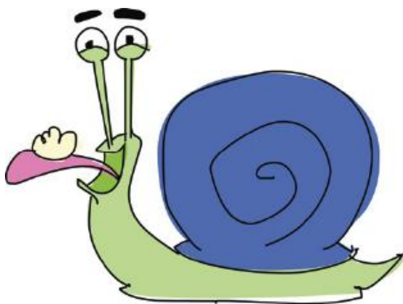
```
>>> x = 0
>>> print('ciao %s' % x)
ciao 0
>>> x = 1
>>> print('ciao %s' % x)
ciao 1
>>> x = 2
>>> print('ciao %s' % x)
ciao 2
>>> x = 3
>>> print('ciao %s' % x)
ciao 3
>>> x = 4
>>> print('ciao %s' % x)
ciao 4
```

L'uso del ciclo perciò ci ha evitato di scrivere otto righe ulteriori di codice. I buoni programmatori odiano fare le cose più di una volta, perciò il ciclo `for` è uno degli enunciati più utilizzati in un linguaggio di programmazione.

Non è necessario usare solo le funzioni `range` e `list` per creare cicli `for`. Potete usare anche una lista già creata, come la lista della spesa del Capitolo 3, in questo modo:

```
>>> mago_lista = ['zampe di ragno', 'dito di rana', 'lingua di
chiocciola', 'ala di pipistrello', 'bava di lumacone', 'rutto di
orso']
>>> for i in mago_lista:
        print(i)
zampe di ragno
dito di rana
lingua di chiocciola
ala di pipistrello
bava di lumacone
rutto di orso
```

Questo codice significa "Per ogni elemento presente nella `mago_lista`, memorizza quel valore nella variabile `i`, poi stampa i contenuti di quella variabile". Anche in questo caso, se volessimo fare a meno del ciclo `for`, dovremmo fare qualcosa di questo genere:



```
>>> mago_lista = ['zampe di ragno', 'dito di rana', 'lingua di
chiocciola', 'ala di pipistrello', 'bava di lumacone', 'rutto di
orso']
>>> print(mago_lista[0])
zampe di ragno
>>> print(mago_lista[1])
dito di rana
>>> print(mago_lista[2])
lingua di chiocciola
>>> print(mago_lista[3])
ala di pipistrello
>>> print(mago_lista[4])
bava di lumacone
>>> print(mago_lista[5])
rutto di orso
```

Ancora una volta, quindi, il ciclo ci ha risparmiato un bel po' di lavoro.

Proviamo a creare un altro ciclo. Scrivete il codice seguente nella shell (il rientro sarà automatico):

```
❶ >>> grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
❷ >>> for i in grandipantalonipelosi:
❸         print(i)
❹         print(i)
❺
❻ grandi
  grandi
  pantaloni
  pantaloni
  pelosi
  pelosi
```

Nella prima riga ❶, abbiamo creato una lista contenente le parole `'grandi'`, `'pantaloni'` e `'pelosi'`. Nella riga successiva, ❷, abbiamo impostato un ciclo che percorre gli elementi della lista, assegnando ciascun elemento alla variabile `i`. Poi abbiamo stampato due volte i contenuti della variabile, nelle due righe successive (❸ e ❹). Premendo INVIO sulla successiva riga vuota ❺, abbiamo detto a Python che il blocco era completo; il codice è stato eseguito e ciascun elemento della lista è stato stampato due volte in ❻.

Ricordate che, se inserite un numero errato di spazi a inizio riga, vi ritroverete con un messaggio di errore. Se aveste inserito il codice precedente con uno spazio in più sulla quarta riga ❹, Python avrebbe visualizzato un errore di rientro (*unexpected indent*):

```
>>> grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
>>> for i in grandipantalonipelosi:
        print(i)
         print(i)

SyntaxError: unexpected indent
```

Come abbiamo visto nel Capitolo 5, Python si aspetta che il numero degli spazi in un blocco sia sempre lo stesso. Non importa quanti spazi inserite, purché siano sempre nello stesso numero in ogni riga (il che rende anche più facile leggere il codice, per gli esseri umani).

Ecco un esempio più complicato di un ciclo `for` con due blocchi di codice:

```
>>> grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
>>> for i in grandipantalonipelosi:
        print(i)
        for j in grandipantalonipelosi:
            print(j)
```

Dove sono i blocchi in questo codice? Il primo blocco è il primo ciclo `for`:

```
grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
for i in grandipantalonipelosi:
    print(i)                        #
    for j in grandipantalonipelosi: # Righe nel PRIMO blocco
        print(j)                    #
```

Il secondo blocco è costituito dalla singola riga `print` nel secondo ciclo `for`:

```
❶ grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
  for i in grandipantalonipelosi:
      print(i)
❷     for j in grandipantalonipelosi:
❸         print(j)      # Questa riga è anche il SECONDO blocco
```

Riuscite a immaginare che cosa combinerà questo piccolo frammento di codice?

Dopo che è stata creata una lista con il nome `grandipantalonipelosi` in ❶, dalle due righe successive possiamo capire che Python percorrerà in ciclo gli elementi della lista e stamperà ciascuno di essi. In ❷, però, deve effettuare un altro ciclo attraverso la lista, questa volta assegnando il valore alla variabile `j`, poi stamperà di nuovo ciascun elemento, come previsto dalla riga ❸. Il codice in ❷ e ❸ è sempre parte del primo ciclo `for`, il che significa che verrà eseguito per ciascun elemento della lista,

nel procedere del primo ciclo `for` basato sulla variabile `i`.

Quando il codice viene eseguito, quindi, dovremmo vedere `grandi` seguito da `grandi`, `pantaloni`, `pelosi` e poi `pantaloni` seguito da `grandi`, `pantaloni`, `pelosi` e così via.

Inserite il codice nella shell di Python e vedetelo con i vostri occhi:

```
>>> grandipantalonipelosi = ['grandi', 'pantaloni', 'pelosi']
>>> for i in grandipantalonipelosi:
❶       print(i)
        for j in grandipantalonipelosi:
❷               print(j)

◈ grandi
  grandi
  pantaloni
  pelosi
◈ pantaloni
  grandi
  pantaloni
  pelosi
◈ pelosi
  grandi
  pantaloni
  pelosi
```

Python entra nel primo ciclo e stampa un elemento della lista in ❶. Poi entra nel secondo ciclo e stampa tutti gli elementi della lista in ❷. Poi continua con il comando `print(i)`, stampando l'elemento successivo della lista, quindi stampa di nuovo tutta la lista con `print(j)`. Nell'output, le righe evidenziate con il simbolo ◈ vengono stampate dall'enunciato `print(i)`, le altre da `print(j)`.

Che ne dite di qualcosa di più utile che stampare parole stupide? Ricordate il calcolo che abbiamo fatto nel Capitolo 2 per sapere quante monete d'oro avreste avuto alla fine dell'anno grazie alla folle invenzione del nonno che duplica monete? Era questo:

```
>>> 20 + 10 * 365 - 3 * 52
```

Ovvero: 20 monete trovate, più 10 monete magiche al giorno per 365 giorni in un anno, meno 3 monete alla settimana rubate dal corvo.

Potrebbe essere utile vedere come il vostro mucchietto di monete d'oro aumenti ogni settimana. Possiamo farlo con un altro ciclo `for`, ma prima dobbiamo cambiare il valore delle nostre `monete_magiche`, in modo che rappresenti il numero totale di monete magiche per settimana, cioè 10 monete magiche al giorno per 7 giorni alla settimana. Quindi `monete_magiche` deve essere 70:

```
>>> monete_trovate = 20
>>> monete_magiche = 70
>>> monete_rubate = 3
```

Possiamo vedere crescere il nostro tesoro ogni settimana creando un’altra variabile, che possiamo chiamare `monete`, e usando un ciclo:

```
>>> monete_trovate = 20
>>> monete_magiche = 70
>>> monete_rubate = 3
❶ >>> monete = monete_trovate
❷ >>> for settimana in range(1, 53):
❸             monete = monete + monete_magiche - monete_rubate
❹             print('settimana %s = %s' % (settimana, monete))
```

In ❶, alla variabile `monete` viene assegnato il valore della variabile `monete_trovate`: questo è il numero di partenza. La riga successiva, ❷, imposta il ciclo `for`, che eseguirà i comandi nel blocco (costituito dalle righe ❸ e ❹). Ogni volta che viene eseguito il ciclo, la variabile assume come valore il numero successivo nell’intervallo fra 1 e 52.

La riga in ❸ è un po’ più complicata. Fondamentalmente, ogni settimana vogliamo aggiungere il numero delle monete che abbiamo creato magicamente e sottrarre il numero delle monete rubate dal corvo. Potete pensare la variabile `monete` come il vostro forziere. Ogni settimana, le nuove monete vanno a finire nel forziere. Quindi questa riga significa “Sostituisci i contenuti della variabile `monete` con il numero delle monete che avevo, più quelle che ho creato questa settimana”. Fondamentalmente, il segno `(=)` è una sorta di comando, che dice “Fai tutto quello che ti viene chiesto a destra, poi salvalo per il seguito, usando il nome qui a sinistra”.

La riga ❹ è un enunciato `print` che usa dei segnaposto e stampa il numero della settimana e il numero totale delle monete accumulate fino a quel momento. (Se tutto questo non ha molto senso per voi, rileggete “Incorporare valori nelle stringhe” a pagina 30.) Se eseguite il programma, vedrete questo risultato:

```
Python 3.5.1 Shell
File Edit Shell Debug Options Window Help
Python 3.5.1 (v3.5.1:37a07cee5969, Dec  6 2015, 01:38:48) [MSC v.1900 32 bit (In
tel)] on win32
Type "copyright", "credits" or "license()" for more information.
>>>
>>> monete_trovate = 20
>>> monete_magiche = 70
>>> monete_rubate = 3
>>> monete = monete_trovate
>>> for settimana in range(1, 53):
        monete = monete + monete_magiche - monete_rubate
        print('Settimana %s = %s' % (settimana, monete))


Settimana 1 = 87
Settimana 2 = 154
Settimana 3 = 221
Settimana 4 = 288
Settimana 5 = 355
Settimana 6 = 422
Settimana 7 = 489
Settimana 8 = 556
Settimana 9 = 623
Settimana 10 = 690
Settimana 11 = 757
Settimana 12 = 824
Settimana 13 = 891
Settimana 14 = 958
Settimana 15 = 1025
Settimana 16 = 1092
Settimana 17 = 1159
Settimana 18 = 1226
Ln: 65 Col: 4
```

A PROPOSITO DI CICLI…

Un ciclo `for` non è l'unico tipo di ciclo che si può creare in Python. Esiste anche il ciclo `while`. Un ciclo `for` è un ciclo di lunghezza specifica, mentre `while` è un ciclo usato quando non si sa in anticipo a che punto la ripetizione deve terminare.

Immaginatevi una scala con 20 gradini: è dentro casa e sapete di poter salire facilmente. Un ciclo `for` è di questo tipo.

```
>>> for gradino in range (0, 20):
        print(gradino)
```

Ora immaginate una scala che sale lungo una montagna. La montagna è alta, e potreste esaurire le forze prima di raggiungere la cima, o il tempo potrebbe mettersi al brutto, costringendovi a fermarvi. Questa è l'idea del ciclo `while`.

```
>>> gradino = 0
>>> while gradino < 10000:
    print(gradino)
    if stanco == True:
        break
    elif cattivotempo == True:
        break
    else:
        gradino = gradino + 1
```

Se provate a inserire e a eseguire questo codice, otterrete un messaggio di errore. Perché? Perché non abbiamo creato le variabili `stanco` e `cattivotempo`. Anche se qui non c'è abbastanza codice per farne un programma funzionante, questo è un esempio elementare di un ciclo `while`.

Iniziamo creando una variabile chiamata `gradino` con `gradino = 0`. Poi, creiamo un ciclo `while` che controlla se il valore della variabile `gradino` è inferiore a 10.000 (`gradino < 10000`), che è il numero totale dei gradini dalla base della montagna fino alla cima. Finché `gradino` è minore di 10.000, Python continuerà a eseguire il resto del codice.

Con `print(gradino)`, stampiamo il valore della variabile e poi controlliamo se `stanco` è `True` con `if stanco == True`. (`True` è un "valore booleano", espressione che chiariremo nel Capitolo 8.) Se lo è, usiamo la parola chiave `break` (equivale a "interrompi") per uscire dal ciclo. La parola chiave `break` è un modo per uscire da un ciclo (in altre parole, interromperlo), e funziona sia per i cicli `while` che per quelli `for`. Qui ha l'effetto di farci uscire dal blocco e di passare agli enunciati che potrebbero comparire dopo la riga `gradino = gradino + 1`.

La riga `elif cattivotempo == True`: controlla se la variabile `cattivotempo` abbia il valore `True`. Se così è, `break` fa uscire dal ciclo. Se né `stanco` né `cattivotempo` sono `True` (caso `else`), sommiamo 1 al valore della variabile `gradino`, con l'enunciato `gradino = gradino + 1`, e il ciclo continua.

Quindi i passi di un ciclo `while` sono i seguenti:

1. Controlla la condizione.
2. Esegui il codice nel blocco.
3. Ripeti.

Più spesso, un ciclo `while` viene creato con una coppia di condizioni, invece che con una sola, come in questo esempio:

```
❶ >>> x = 45
❷ >>> y = 80
❸ >>> while x < 50 and y < 100:
          x = x + 1
          y = y + 1
          print (x, y)
```

Qui abbiamo creato una variabile `x` con il valore 45 in ❶ e una variabile `y` con valore 80 in ❷. Il ciclo controlla le due condizioni in ❸: se cioè `x` è minore di 50 e se `y` è minore di 100.

Finché entrambe le condizioni sono vere, vengono eseguite le righe seguenti, in cui si somma 1 a entrambe le variabili e poi le si stampa. Ecco l'output prodotto da questo codice:

```
46 81
47 82
48 83
49 84
50 85
```

Riuscite a stabilire come funziona?

Iniziamo a contare da 45 per la variabile x e da 80 per la variabile `y`, poi incrementiamo (sommiamo 1) ciascuna variabile ogni volta che viene eseguito il codice nel ciclo. Il ciclo andrà avanti finché `x` è minore di 50 e `y` è minore di 100. Dopo cinque passaggi nel ciclo (ogni volta a ciascuna variabile viene sommato 1), il valore in `x` raggiunge 50. Ora la prima condizione `(x < 50)` non è più vera, perciò Python sa di dover porre termine al ciclo.

Un altro uso comune di un ciclo `while` è la creazione di cicli "quasi eterni". Sono cicli che potrebbero continuare all'infinito, ma in realtà continuano finché non succede qualcosa nel codice che li interrompe. Ecco un esempio:

```
while True:
    qui una gran quantità di codice
    qui una gran quantità di codice
    qui una gran quantità di codice
    if qualche_valore == True:
        break
```

La condizione del ciclo `while` è semplicemente `True`, che è sempre vera, perciò il codice nel blocco continuerà a essere eseguito (in questo senso, il ciclo è eterno). Python esce dal ciclo solo se la variabile `qualche_valore` è vera. Potete vedere un esempio migliore di questo in "Usare randint per scegliere un numero a caso" a pagina 134, ma forse è meglio che aspettiate di aver letto il Capitolo 7.

CHE COSA AVETE IMPARATO

In questo capitolo, abbiamo usato i cicli per eseguire attività ripetitive senza doverci ripetere. Abbiamo detto a Python che cosa volevamo fosse ripetuto scrivendo le relative attività all'interno di blocchi di codice, che abbiamo inserito all'interno di cicli. Abbiamo usato due tipi di cicli: `for` e `while`, che sono simili ma si possono usare in modi diversi. Abbiamo usato anche la parola chiave `break` per interrompere i cicli, cioè per uscire da un ciclo.

ROMPICAPO DI PROGRAMMAZIONE

Ecco qualche esempio di ciclo che potete provare a costruire voi stessi. Le risposte si possono trovare all'indirizzo *http://python-for-kids.com/*.

1. IL CICLO "CIAO"

Che cosa pensate che faccia il codice seguente? Prima provate ad arrivarci ragionando, poi scrivete il codice in Python ed eseguitelo, per vedere se avevate ragione.

```
>>> for x in range(0, 20):
        print('ciao %s' % x)
        if x < 9:
            break
```

2. NUMERI PARI

Create un ciclo che stampi i numeri pari finché non raggiunge il numero corrispondente alla vostra età oppure, se la vostra età è un numero dispari, che stampi i numeri dispari finché non raggiunge la vostra età. Per esempio, potrebbe stampare una cosa come questa:

```
2
4
6
8
10
12
14
```

3. I CINQUE INGREDIENTI PREFERITI

Create una lista che contenga cinque diversi ingredienti per un buon panino, per esempio:

```
>>> ingredienti = ['lumache', 'sanguisughe', 'ombelichi di
gorilla', 'ciglia di bruco', 'zampe di millepiedi']
```

Ora create un ciclo che stampi la lista (compresi i numeri):

```
1 lumache
2 sanguisughe
3 ombelichi di gorilla
4 ciglia di bruco
5 zampe di millepiedi
```

4. IL PESO SULLA LUNA

Se in questo momento vi trovaste sulla Luna, il vostro peso sarebbe pari al 16,5 per cento di quello che è sulla Terra. Potete calcolarlo moltiplicando il vostro peso terrestre per 0,165.

Se il vostro peso aumentasse di un chilogrammo ogni anno nei prossimi 15 anni, quale sarebbe il vostro peso se visitaste la Luna ogni anno e alla fine dei 15 anni? Scrivete un programma che usi un ciclo `for` e stampi il vostro peso lunare per ciascun anno.

7
RICICLARE IL CODICE CON FUNZIONI E MODULI

Pensate quante cose buttate via ogni giorno: bottiglie dell'acqua, lattine, confezioni di patatine, incarti di panini, sacchetti che contenevano filetti di carota o fette di mela, borse della spesa, giornali, riviste e così via. Ora immaginate che cosa succederebbe se tutta quella spazzatura venisse ammonticchiata davanti all'ingresso di casa vostra, senza separare carta, plastica, lattine e vetro.

Ovviamente, probabilmente riciclate il più possibile, che è una buona cosa, perché nessuno vorrebbe scalare una montagna di spazzatura sulla strada per andare a scuola. Anziché starsene in un'enorme pila alla rinfusa, le bottiglie di vetro che riciclate vengono fuse e trasformate in nuovi contenitori e nuove bottiglie; la carta viene macerata e trasformata in carta riciclata; la plastica viene trasformata in altri oggetti di plastica. Così riusiamo cose che altrimenti butteremmo semplicemente via.

Nel mondo della programmazione, il riuso è altrettanto importante. Ovviamente, il vostro programma non scomparirà sotto una montagna di rifiuti, ma se non riusate qualcosa di quello che fate, alla fine le dita vi si ridurranno a moncherini doloranti a furia di scrivere codice. La tecnica del riuso inoltre rende il codice più breve e più facile da leggere.

USARE LE FUNZIONI

Abbiamo già visto uno dei modi per riciclare codice in Python. Nel capitolo precedente, abbiamo usato le funzioni `range` e `list` per far "contare" Python.

```
>>> list(range(0, 5))
[0, 1, 2, 3, 4]
```

Se sapete contare, non è difficile creare una lista di numeri consecutivi scrivendoli direttamente, ma, più lunga è la lista, più dovrete scrivere. Invece, se usate le funzioni, potete creare con la massima facilità anche una lista di migliaia di numeri. Ecco un esempio che usa le funzioni `list` e `range` per produrre una lista di numeri:

```
>>> list(range(0, 1000))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16…, 997, 998, 999]
```

Le *funzioni* sono pezzi di codice che dicono a Python di fare qualcosa. Sono un modo per riusare il codice: potete usare le funzioni molte volte all'interno dei vostri programmi.

Quando scrivete programmi semplici, le funzioni sono comode; quando comincerete a scrivere programmi lunghi e più complessi, come i giochi, le funzioni diventano essenziali (immaginando che vogliate finire di scrivere il vostro programma entro questo secolo).

PARTI DI UNA FUNZIONE

Una funzione ha tre parti: un *nome*, *parametri* e un *corpo*. Ecco un esempio di una funzione semplice:

```
>>> def funzprova(mionome):
        print('ciao %s' % mionome)
```

Il nome di questa funzione è `funzprova`. Ha un unico parametro, `mionome`, e il suo corpo è il blocco di codice che segue immediatamente la riga che inizia con `def` (abbreviazione di *define*, definisci).

Un *parametro* è una variabile che esiste solo quando viene usata una funzione.

Potete eseguire la funzione chiamandone il nome, racchiudendo poi fra parentesi il valore del parametro:

```
>>> funzprova('Maria')
ciao Maria
```

Una funzione può avere due, tre o un numero qualsiasi di parametri, non uno solamente:

```
>>> def funzprova(nome, cognome):
        print('ciao %s %s' % (nome, cognome))
```

I due valori per questi parametri devono essere separati da una virgola:

```
>>> funzprova('Maria', 'Rossi')
ciao Maria Rossi
```

Potremmo anche creare prima delle variabili e chiamare poi la funzione con quelle variabili:

```
>>> nome = 'Giorgio'
>>> cognome = 'Ruberti'
>>> funzprova(nome, cognome)
ciao Giorgio Ruberti
```

Si usa spesso una funzione per avere di ritorno un valore, attraverso un enunciato `return`. Per esempio, potreste scrivere una funzione che calcoli quanti soldi avete risparmiato:

```
>>> def risparmiati(paghetta, mancia, spese):
        return paghetta + mancia - spese
```

Questa funzione prende tre parametri: somma i primi due (`paghetta` e `mancia`) e sottrae l'ultimo (`spese`). Il risultato viene restituito e può essere assegnato a una variabile (analogamente a come abbiamo assegnato valori alle variabili in altri casi) oppure stampato:

```
>>> print(risparmiati(10, 10, 5))
15
```

VARIABILI E AMBITO

Una variabile che si trova all'interno del corpo di una funzione non può essere usata nuovamente, una volta terminata l'esecuzione della funzione, perché esiste solo al suo interno. Nel mondo della programmazione, si dice che quello è il suo ambito (*scope*, in inglese).

Vediamo una funzione che usa un paio di variabili ma non ha alcun parametro:

```
❶ >>> def test_variabile():
          prima_var = 10
          seconda_var = 20
❷         return prima_var * seconda_var
```

In questo esempio, in ❶ abbiamo creato una funzione che si chiama `test_variabile`, che moltiplica due variabili (`prima_var` e `seconda_var`) e restituisce il risultato in ❷.

```
>>> print(test_variabile())
200
```

Se chiamiamo questa funzione utilizzando `print`, otteniamo il risultato: 200. Se però tentiamo di stampare i contenuti di `prima_var` (o di `seconda_var`, poco importa) all'esterno del blocco di codice della funzione, otteniamo un messaggio di errore:

```
>>> print(prima_var)
Traceback (most recent call last):
  File "<pyshell#33>", line 1, in <module>
    print(prima_var)
NameError: name 'prima_var' is not defined
```

Se una variabile è definita all'esterno della funzione, ha un ambito diverso. Per esempio, definiamo una variabile prima di creare la funzione, poi cerchiamo di usarla dentro la funzione:

```
❶ >>> altra_var = 100
  >>> def test_variabile2():
          prima_var = 10
          seconda_var = 20
❷         return prima_var * seconda_var * altra_var
```

In questo caso, anche se le variabili `prima_var` e `seconda_var` non possono essere usate all’esterno della funzione, `altra_var` (che è stata creata esternamente alla funzione in ❶) può essere utilizzata al suo interno, in ❷.

Ecco il risultato che si ottiene chiamando questa funzione:

```
>>> print(test_variabile2())
20000
```

Ora, supponiamo che vogliate costruire una navicella spaziale con qualche materiale economico, per esempio lattine usate. Pensate di poter appiattire 2 lattine a settimana per creare le pareti curve della navicella, ma per finire la fusoliera vi serviranno 500 lattine. Possiamo scrivere facilmente una funzione che ci permetta di calcolare quanto ci vorrà per lavorare 500 lattine, se possiamo lavorare 2 lattine a settimana.

Creiamo una funzione che mostri quante lattine abbiamo appiattito in un anno (52 settimane). La nostra funzione prenderà come parametro il numero delle lattine:

```
>>> def costruzione_navicella(lattine):
        totale_lattine = 0
        for settimana in range(1, 53):
            totale_lattine = totale_lattine + lattine
            print('Settimana %s = %s lattine' % (settimana,
                  totale_lattine))
```

Nella prima riga della funzione, abbiamo creato una variabile, `totale_lattine`, e abbiamo impostato il suo valore a 0. Poi abbiamo creato un ciclo per le settimane dell’anno e sommato ogni settimana il numero delle lattine lavorate. Questo blocco di codice costituisce il contenuto della nostra funzione. In questa funzione però c’è anche un altro blocco di codice: le ultime due righe, che sono il blocco del ciclo `for`.

Proviamo a inserire questa funzione nella shell e a chiamarla con valori diversi per il numero delle lattine:

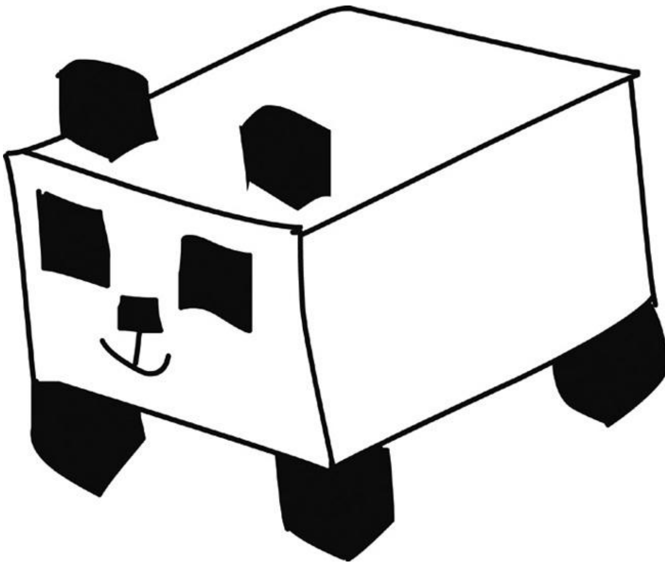
```
>>> costruzione_navicella(2)
Settimana 1 = 2 lattine
Settimana 2 = 4 lattine
Settimana 3 = 6 lattine
Settimana 4 = 8 lattine
Settimana 5 = 10 lattine
Settimana 6 = 12 lattine
Settimana 7 = 14 lattine
Settimana 8 = 16 lattine
Settimana 9 = 18 lattine
Settimana 10 = 20 lattine
(continua…)

>>> costruzione_navicella(13)
Settimana 1 = 13 lattine
Settimana 2 = 26 lattine
Settimana 3 = 39 lattine
Settimana 4 = 52 lattine
Settimana 5 = 65 lattine
(continua…)
```

Questa funzione può essere riutilizzata con valori diversi per il numero delle lattine lavorate ogni settimana, il che è un po' più efficiente che dover riscrivere il ciclo `for` ogni volta che si vuole provare la funzione con un numero diverso.

Le funzioni possono anche essere raggruppate in moduli: ed è qui che Python diventa davvero molto utile.

USARE I MODULI

I `moduli` raggruppano funzioni, variabili e altro ancora, utilizzabili per scrivere programmi più grandi e più potenti. Alcuni moduli sono già incorporati in Python; altri si possono scaricare separatamente. Vedrete bene l'utilità dei moduli (come `tkinter`, incorporato, e `PyGame`, che invece non lo è) quando scriverete dei giochi, o per la manipolazione delle immagini (per esempio `PIL`, Python Imaging Library) o per creare grafica tridimensionale (per esempio `Panda3D`).



Si possono usare moduli per moltissime cose utili. Per esempio, se progettaate un gioco di simulazione, e desiderate che il mondo del gioco cambi realisticamente, potreste calcolare la data e l'ora correnti utilizzando un modulo incorporato che si chiama `time`:

```
>>> import time
```

In questo caso si usa il comando `import` per dire a Python che si vuole usare il modulo `time`. Poi, si possono chiamare le funzioni presenti in questo modulo, utilizzando il simbolo del punto. (Ricordate che abbiamo usato funzioni di questo genere per lavorare con il modulo `turtle` nel Capitolo 4, per esempio `t.forward(50)`.) Per esempio, ecco come potremmo chiamare la funzione `asctime` con il modulo `time`:

```
>>> print(time.asctime())
'Tue May 24 11:17:39 2016'
```

La funzione `asctime` fa parte del modulo `time` e restituisce la data e l'ora correnti, sotto forma di stringa (in inglese: *Tue* sta per *Tuesday*, martedì, *May* è maggio, in questo esempio).

Ora supponiamo che vogliate chiedere a qualcuno che usa il vostro programma di inserire un valore, magari la data di nascita o l'età. Potete farlo utilizzando un enunciato `print` per visualizzare un messaggio, e il modulo `sys` (abbreviazione di *system*, sistema), che contiene funzioni di servizio per interagire con il sistema Python stesso. Prima di tutto, importiamo il modulo `sys`:

```
>>> import sys
```

All’interno del modulo `sys` si trova un *oggetto* speciale chiamato `stdin` (sta per *standard input*), che mette a disposizione una funzione molto utile, `readline`. Questa funzione viene utilizzata per leggere (*read*) una riga di testo scritta alla tastiera fino alla pressione del tasto INVIO. (Vedremo come funzionano gli oggetti nel Capitolo 8.) Per provare `readline`, inserite questo codice nella shell:

```
>>> import sys
>>> print(sys.stdin.readline())
```

Se poi scrivete qualche parola e premete INVIO, quelle parole verranno stampate nella shell.

Ripensate al codice che abbiamo scritto nel Capitolo 5, utilizzando un enunciato `if`.

Anziché creare la variabile `età` e attribuirle un valore specifico prima dell’enunciato `if`, ora possiamo chiedere a qualcuno di inserire quel valore. Prima, però, trasformiamo quel codice in una funzione:

```
>>> def battuta_stupida(età):
        if età >= 10 and età <= 13:
            print('Cosa fa 13 + 49 + 84 + 155 + 98? Emicrania!')
        else:
            print('Cosa?')
```

Ora potete chiamare la funzione scrivendo il suo nome e poi dicendole che numero usare inserendo quel numero fra parentesi. Funziona?

```
>>> battuta_stupida(9)
Cosa?
>>> battuta_stupida(10)
Cosa fa 13 + 49 + 84 + 155 + 98? Emicrania!
```

Funziona! Ora facciamo in modo che la funzione chieda l’età della persona. (Potete modificare o aggiungere qualcosa a una funzione tutte le volte che volete.)

```
>>> def battuta_stupida():
        print('Quanti anni hai?')
❶       età = int(sys.stdin.readline())
❷       if età >= 10 and età <= 13:
            print('Cosa fa 13 + 49 + 84 + 155 + 98? Emicrania!')
        else:
            print('Cosa?')
```

Avete riconosciuto la funzione `int` nella riga ❶, che converte una stringa in un numero? L'abbiamo usata perché `readline()` restituisce tutto quello che viene scritto alla tastiera sotto forma di stringa, mentre noi vogliamo un numero, per poterlo confrontare con i numeri 10 e 13 nella riga ❷. Per provare, chiamate la funzione senza parametri, poi scrivete un numero quando appare la richiesta `Quanti anni hai?`:

```
>>> battuta_stupida()
Quanti anni hai?
10
Cosa fa 13 + 49 + 84 + 155 + 98? Emicrania!
>>> battuta_stupida()
Quanti anni hai?
15
Cosa?
```

CHE COSA AVETE IMPARATO

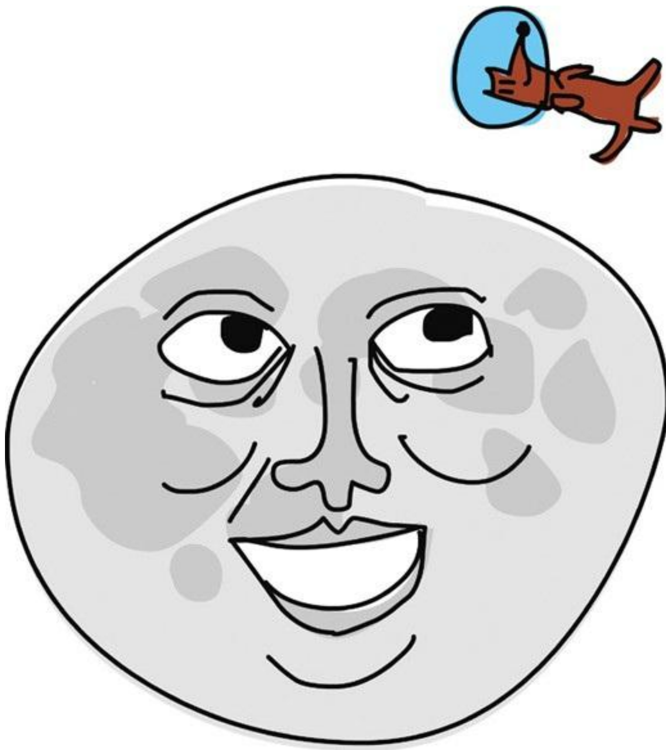
In questo capitolo, avete visto come creare pezzi di codice riusabile in Python con le funzioni e come usare le funzioni messe a disposizione dai moduli. Avete imparato che l'ambito delle variabili controlla se possono essere viste all'interno o all'esterno delle funzioni, e come creare funzioni con la parola chiave `def`. Avete anche scoperto come importare moduli, per poterne utilizzare i contenuti.

ROMPICAPO DI PROGRAMMAZIONE

Provate gli esempi seguenti, per sperimentare come si creano funzioni. Le risposte si possono trovare all'indirizzo *http://python-for-kids.com*.

1. FUNZIONE PER IL PESO SULLA LUNA

Nel Capitolo 6, uno dei rompicapo era come creare un ciclo `for` per determinare il vostro peso sulla Luna nell'arco di 15 anni. Quel ciclo `for` può essere trasformato facilmente in una funzione. Provate a creare una funzione che prenda un peso iniziale e lo aumenti ogni anno. Potete chiamare la nuova funzione per esempio in questo modo:



```
>>> peso_lunare(30, 0.25)
```

2. FUNZIONE PER IL PESO SULLA LUNA E GLI ANNI

Prendete la funzione appena creata e modificata in modo che permetta di stabilire il peso per periodi diversi, per esempio 5 o 20 anni. Fate attenzione a modificare la funzione in modo che prenda tre argomenti: il peso iniziale, il peso di cui si aumenta ogni anno e il numero degli anni:

```
>>> peso_lunare(90, 0.25, 5)
```

3. PROGRAMMA PER IL PESO SULLA LUNA

Anziché una semplice funzione, a cui passare i valori come parametri, potete creare un piccolo programma che richieda i valori all'utente mediante `sys.stdin.readline()`. In questo caso, chiamerete la funzione senza alcun parametro:

```
>>> peso_lunare()
```

La funzione presenterà un messaggio in cui vi chiede il peso di partenza, un secondo messaggio che vi chiede di quanto aumenta il peso ogni anno, infine un messaggio che vi chiede il numero degli anni. Dovreste poter vedere qualcosa di questo tipo:

```
Per favore inserisci il tuo peso terrestre attuale
45
Per favore inserisci di quanto eumenterà il tuo peso ogni anno
0.4
Per favore inserisci il numero degli anni
12
```

Ricordate di importare il modulo sys prima di creare la vostra funzione:

```
>>> import sys
```

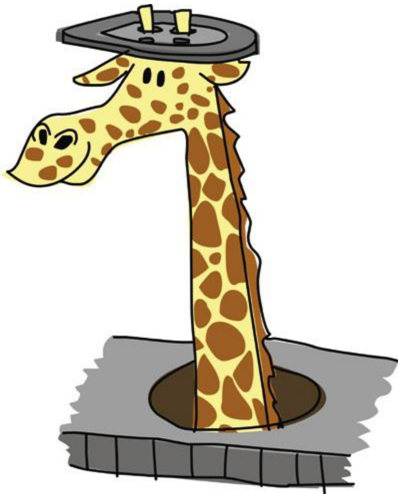
8
COME SI USANO CLASSI E OGGETTI

Perché una giraffa è come un marciapiedi?
Perché entrambi sono *cose*, quelli che in italiano si definiscono *nomi* e in Python *oggetti*.
L'idea di *oggetto* è importante nel mondo informatico. Gli oggetti sono un modo per organizzare il codice in un programma e suddividere le cose, in modo da semplificare il ragionamento su idee complesse. (Abbiamo già usato un oggetto nel Capitolo 4, quando abbiamo lavorato con la tartaruga: era `Pen`.)

Per capire esattamente come funzionano gli oggetti in Python, dobbiamo pensare ai tipi di oggetti. Cominciamo con le giraffe e i marciapiedi.

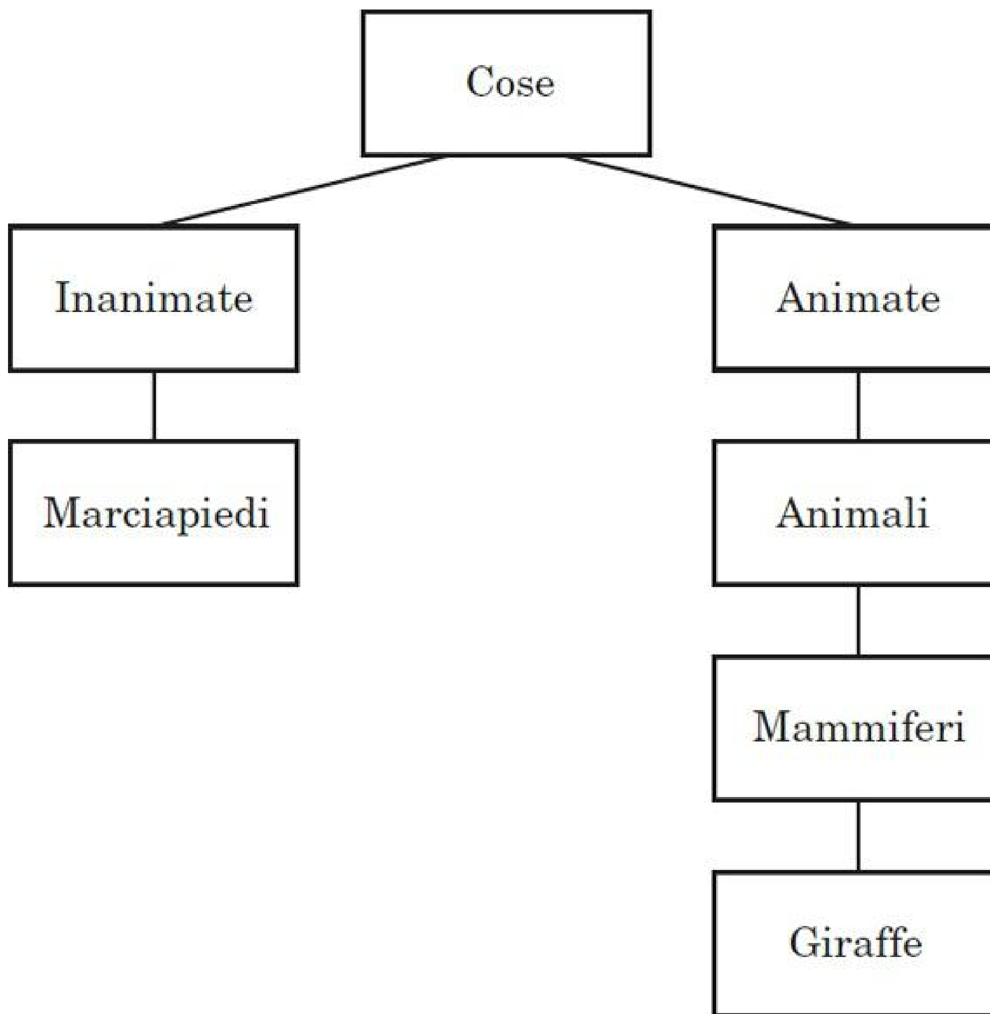
Una giraffa è un tipo di mammifero, che è un tipo di animale. Una giraffa è anche una cosa animata (è viva).

Ora prendiamo un marciapiede. Non c'è molto da dire, se non che non è una cosa vivente. Chiamiamolo una cosa inanimata (cioè non è vivo). I termini *mammifero*, *animale*, *animata* e *inanimata* sono tutti termini per classificare le cose.



SUDDIVIDERE LE COSE IN CLASSI

In Python, gli oggetti sono definiti da *classi*, che possiamo pensare come un modo per classificare gli oggetti in gruppi. Ecco un diagramma ad albero delle classi a cui apparterrebbero giraffe e marciapiedi, in base alle nostre definizioni precedenti:



La classe principale è `Cose`. Sotto la classe `Cose` abbiamo `Inanimate` e `Animate`. Queste sono a loro volta articolate in `Marciapiedi` per `Inanimate`, `Animali`, `Mammiferi` e `Giraffe` per `Animate`.

Possiamo usare le classi per organizzare pezzi di codice di Python. Per esempio, prendiamo il modulo `turtle`. Tutte le cose che può fare questo modulo (per esempio andare avanti, indietro, girare a sinistra, a destra) sono funzioni contenute nella classe `Pen`. Si più pensare un oggetto come membro di una classe, e si può creare qualsiasi numero di oggetti per una classe – come faremo fra breve.

Ora creiamo lo stesso insieme di classi rappresentato nel diagramma ad albero, partendo dalla cima. Le classi si definiscono usando la parola chiave `class` seguita da un nome. Poiché `Cose` è la classe più ampia, partiamo da questa:

```
>>> class Cose:
        pass
```

Chiamiamo `Cose` questa classe, poi usiamo l'enunciato `pass` per far sapere a Python che non gli forniremo ulteriori informazioni: si usa `pass` quando si vuol creare una classe o una funzione, ma non si vogliono specificare per il momento i suoi dettagli.

Poi, aggiungeremo le altre classi e costruire alcune relazioni fra di esse.

FIGLI E GENITORI

Se una classe è contenuta in un'altra classe, si dice che è *figlia* di quella classe (che sarà la sua *genitrice*). Una classe può essere figlia di qualche classe e genitrice di altre. Nel nostro diagramma, la classe che sta sopra una certa classe è la sua genitrice. Per esempio, `Inanimate` e `Animate` sono entrambe figlie della classe `Cose`, che quindi è la loro genitrice.

Per dire a Python che una classe è figlia di un'altra, aggiungiamo il nome della classe genitrice fra parentesi dopo il nome della nuova classe, in questo modo:

```
>>> class Inanimate(Cose):
        pass
>>> class Animate(Cose):
        pass
```

Abbiamo creato una classe, chiamata `Inanimate`, e abbiamo detto a Python che la sua genitrice è `Cose`, mediante il codice `class Inanimate(Cose)`. Poi abbiamo creato una classe chiamata `Animate` e abbiamo detto che anche in questo caso la genitrice è `Cose`, mediante il codice `class Animate(Cose)`. Proviamo a fare la stessa cosa con la classe `Marciapiede`, figlia di `Inanimate`:

```
>>> class Marciapiede(Inanimate):
        pass
```

Possiamo poi analogamente organizzare le classi `Animali`, `Mammiferi` e `Giraffe`, utilizzando le relative classi genitrici:

```
>>> class Animali(animate):
        pass
>>> class Mammiferi(Animali):
        pass
>>> class Giraffe(Mammiferi):
        pass
```

AGGIUNGERE OGGETTI ALLE CLASSI

Ora che abbiamo una serie di classi, che ne dite di inserire un po' di cose dentro quelle classi? Supponiamo di avere una giraffa che si chiama Reginald. Sappiamo che appartiene alla classe `Giraffe`, ma che cosa possiamo usare, in termini di programmazione, per descrivere quella giraffa che si chiama Reginald? Diciamo che Reginald è un oggetto della classe `Giraffe` (si dice anche a volte che è una istanza della classe). Per "presentare" Reginald a Python, usiamo questa forma:

```
>>> reginald = Giraffe()
```

Questa riga di codice dice a Python di creare un oggetto nella classe `Giraffe` e di assegnarlo alla variabile `reginald`. Come nel caso di una funzione, il nome della classe è seguito dalle due parentesi. Nel seguito del capitolo vedremo come creare

oggetti utilizzando dei parametri fra le parentesi.

Che cosa fa l'oggetto `reginald`? Beh, niente, per il momento. Per rendere utili i nostri oggetti, quando creiamo le nostre classi, dobbiamo definire anche delle funzioni che possano essere usate con gli oggetti di quella classe. Anziché usare la parola chiave `pass` subito dopo la definizione della classe, possiamo aggiungere definizioni di funzioni.

DEFINIRE FUNZIONI DI CLASSI

Nel Capitolo 7 abbiamo presentato le funzioni come un modo per riusare codice. Una funzione associata a una classe si definisce come qualsiasi altra funzione, solo che si scrive rientrata sotto la definizione della classe. Per esempio, ecco una funzione normale che non è associata a una classe:

```
>>> def una_funzione_normale():
        print('Io sono una funzione normale')
```

Ecco un paio di funzioni che appartengono a una classe:

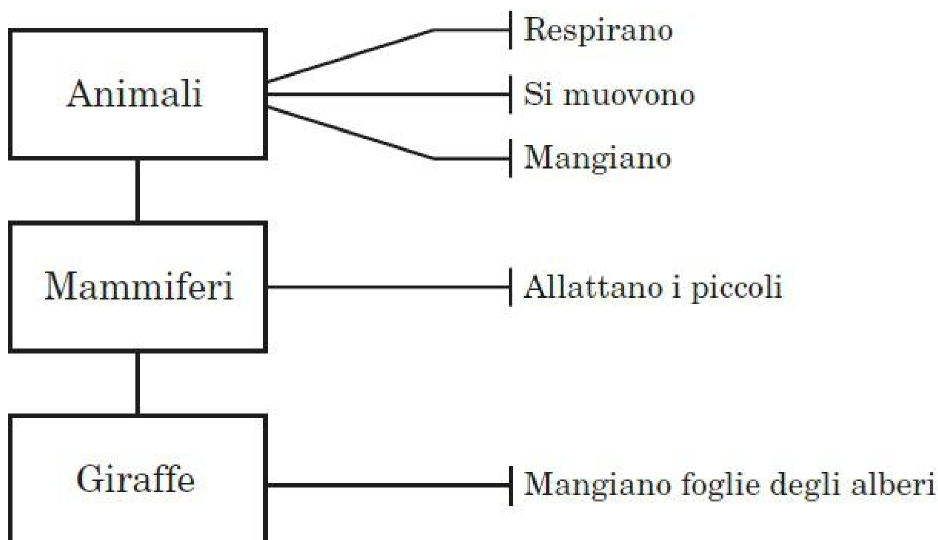
```
>>> class ClasseBanale:
        def una_funzione_di_classe():
            print('Io sono una funzione di classe')
        def altra_funzione_di_classe():
            print("Anch'io sono una funzione di classe, vedi?")
```

CARATTERISTICHE COME FUNZIONI

Considerate le classi figlie della classe `Animate` definita a p. 95. Possiamo aggiungere *caratteristiche* a una classe per dire che cos'è e che cosa può fare. Una caratteristica è un tratto comune a tutti i membri della classe (e di tutte le classi figlie).

Per esempio, che cos'hanno in comune tutti gli animali? Tanto per cominciare, tutti respirano. Poi, tutti si muovono e mangiano. E i mammiferi? I mammiferi allattano i piccoli. E respirano, si muovono e mangiano. Sappiamo che le giraffe mangiano le foglie degli alberi e, come tutti i mammiferi, allattano, respirano, si muovono e mangiano.

Aggiungiamo al nostro diagramma queste caratteristiche:



Queste caratteristiche possono essere considerate azioni, o *funzioni* – cose che un oggetto di quella classe può fare.

Per aggiungere una funzione a una classe, si usa la parola chiave `def`. La classe `Animali` si potrebbe definire quindi così:

```
>>> class Animali(Animate):
        def respira(self):
            pass
        def si_muove(self):
            pass
        def mangia(self):
            pass
```

Nella prima riga, definiamo la classe come in precedenza ma, invece di usare la parola chiave `pass` sulla riga successiva, definiamo una funzione `respira` e le assegniamo un parametro: `self` (letteralmente significa "sé"). Il parametro `self` è un modo per consentire a una funzione della classe di chiamare un'altra funzione nella stessa classe (e nella classe genitrice). Vedremo più avanti come viene usato questo parametro.

Sulla riga successiva, `pass` dice a Python che per il momento non gli forniremo altre informazioni sulla funzione `respira`, perché per ora non dovrà fare nulla. Poi aggiungiamo le funzioni `si_muove e mangia`, e anche queste per ora non fanno nulla. Ricreeremo le classi fra breve e inseriremo codice utile nelle funzioni. Questo è un modo molto diffuso di sviluppare i programmi: spesso i programmatori creano classi con funzioni che non fanno nulla, per stabilire che cosa le classi debbano fare, prima di entrare nei dettagli delle singole funzioni.

Possiamo aggiungere funzioni anche alle altre due classi, `Mammiferi` e `Giraffe`. Ciascuna classe sarà in grado di usare le caratteristiche (cioè le funzioni) della genitrice. Questo significa che non è necessario creare classi molto complicate: potete inserire le vostre funzioni nella classe più generale per cui vale quella caratteristica, e tutte le sue figlie la potranno usare. (Questo è un buon modo per rendere le classi più semplici e più facili da comprendere.)

```
>>> class Mammiferi(Animali):
        def allatta(self):
            pass
>>> class Giraffe(Mammiferi):
        def mangia_foglie_degli_alberi(self):
            pass
```

PERCHÉ USARE CLASSI E OGGETTI?

Abbiamo aggiunto funzioni alle nostre classi, ma perché usare classi e oggetti, quando si potrebbero semplicemente scrivere funzioni normali chiamate `respira`, `si_muove`, `mangia` e così via?

Per rispondere, useremo la nostra giraffa Reginald, creata come oggetto della classe `Giraffe`, in questo modo:

```
>>> reginald = Giraffe()
```

Poiché `reginald` è un oggetto possiamo chiamare (eseguire) le funzioni fornite dalla sua classe (`Giraffe`) e dalle antenate di questa. Si chiamano le funzioni su un oggetto usando l'operatore punto e il nome della funzione. Per dire a Reginald di muoversi o mangiare, possiamo chiamare le funzioni così:

```
>>> reginald = Giraffe()
>>> reginald.si_muove()
>>> reginald.mangia_foglie_degli_alberi()
```

Supponiamo che Reginald abbia un amico (sempre una giraffa) che si chiama Harold. Creiamo un altro oggetto della classe `Giraffe` con il nome `harold`:

```
>>> harold = Giraffe()
```

Poiché usiamo oggetti e classi, possiamo dire a Python esattamente di quale giraffa stiamo parlando quando vogliamo eseguire la funzione `si_muove`. Per esempio, se volessimo far muovere Harold, ma lasciare fermo Reginald, potremmo chiamare `si_muove` utilizzando l'oggetto `harold`, così:

```
>>> harold.si_muove()
```

In questo caso, si muoverà solo Harold.

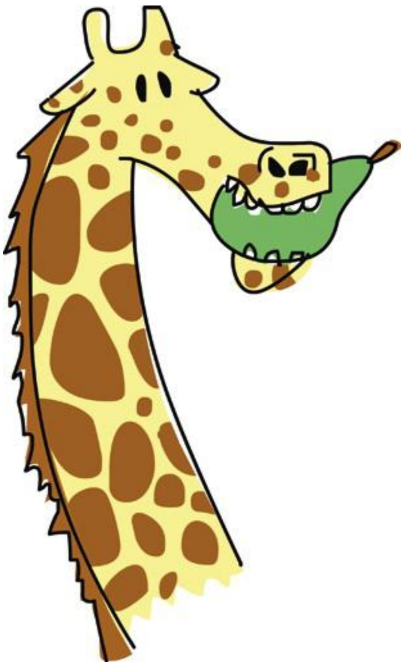
Modifichiamo un po' le classi per renderlo più evidente. Aggiungiamo alle funzioni un enunciato `print`, invece di `pass`:

```
>>> class Animali(Animate):
        def respira(self):
                print('respira')
        def si_muove(self):
                print('si muove')
        def mangia(self):
                print('mangia')
>>> class Mammiferi(Animali):
        def allatta(self):
                print('allatta')
>>> class Giraffe(Mammiferi):
        def mangia_foglie_degli_alberi(self):
                print('mangia foglie degli alberi')
```

Ora, quando creiamo `reginald` e `harold` e chiamiamo delle funzioni su di essi, vediamo succedere qualcosa di concreto:

```
>>> reginald = Giraffe()
>>> harold = Giraffe()
>>> reginald.si_muove()
si muove
>>> harold.mangia_foglie_degli_alberi()
mangia foglie degli alberi
```

Nelle prime due righe, abbiamo creato le variabili `reginald` e `harold`, che sono oggetti della classe `Giraffe`. Poi, abbiamo chiamato la funzione `si_muove` per `reginald`, e Python ha stampato `si muove` sulla riga successiva. Analogamente, abbiamo chiamato la funzione `mangia_foglie_degli_alberi` per `harold` e Python ha stampato `mangia foglie degli alberi`. Se si trattasse di giraffe vere, anziché di oggetti in un computer, una giraffa starebbe camminando, l’altra starebbe mangiando.



OGGETTI E CLASSI NELLE IMMAGINI

E se usassimo un approccio più grafico a oggetti e classi? Torniamo al modulo

turtle con cui abbiamo giocato nel Capitolo 4. Quando usiamo turtle.Pen(), Python crea un oggetto della classe Pen che è fornito dal modulo turtle (assomiglia un po' ai nostri oggetti reginald e harold della sezione precedente). Possiamo creare due oggetti tartaruga (Avery e Kate), esattamente come abbiamo creato due giraffe:

```
>>> import turtle
>>> avery = turtle.Pen()
>>> kate = turtle.Pen()
```

Ciascun oggetto tartaruga (avery e kate) è membro della classe Pen.

Qui gli oggetti cominciano a dimostrare la loro potenza. Avendo creato i nostri oggetti tartaruga, possiamo chiamare le funzioni per ciascuno di essi, e disegneranno in modo indipendente l'uno dall'altro. Provate con queste istruzioni:

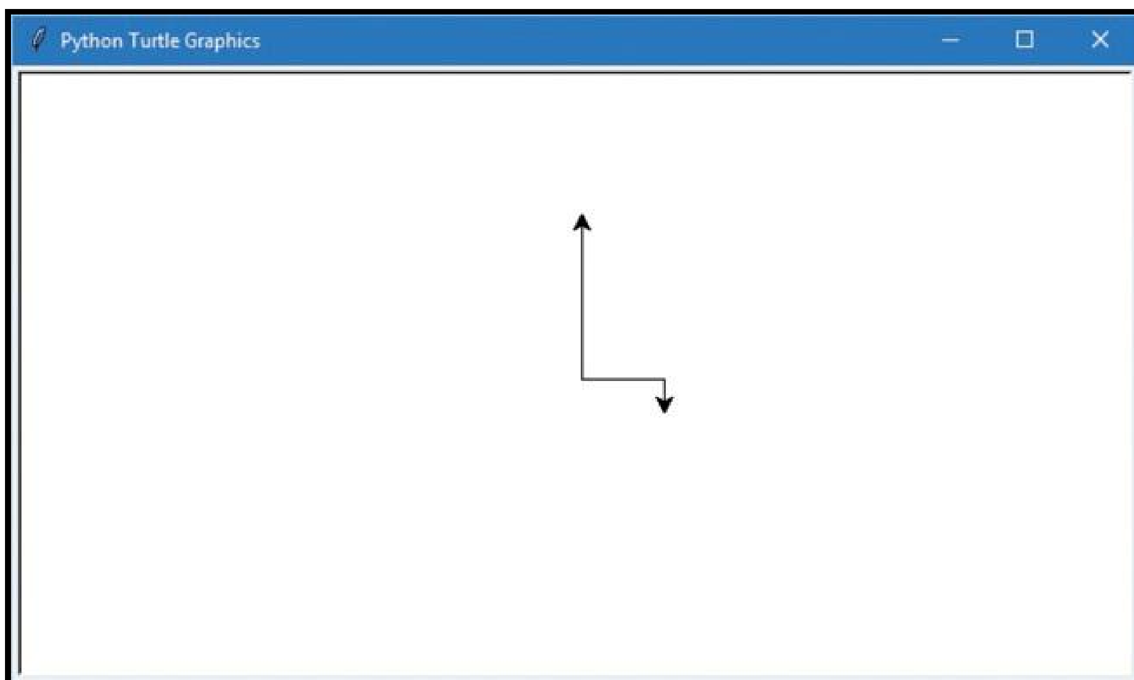
```
>>> avery.forward(50)
>>> avery.right(90)
>>> avery.forward(20)
```

Con questa serie di istruzioni, diciamo ad Avery di procedere in avanti per 50 pixel, poi di girare a destra di 90 gradi e di procedere in avanti per 20 pixel, in modo che si arresti con la punta verso il basso. Ricordate che le tartarughe inizialmente sono sempre orientate verso destra.

Ora è il momento di spostare Kate.

Così diciamo a Kate di ruotare verso sinistra di 90 gradi, poi di andare avanti per 100 pixel: così si fermerà con la punta rivolta verso l'alto.

Fin qui, abbiamo ottenuto una linea con le frecce che si muovevano in direzioni diverse, dove ciascuna freccia rappresenta un diverso oggetto tartaruga: Avery che punta verso il basso, Kate che punta verso l'alto.

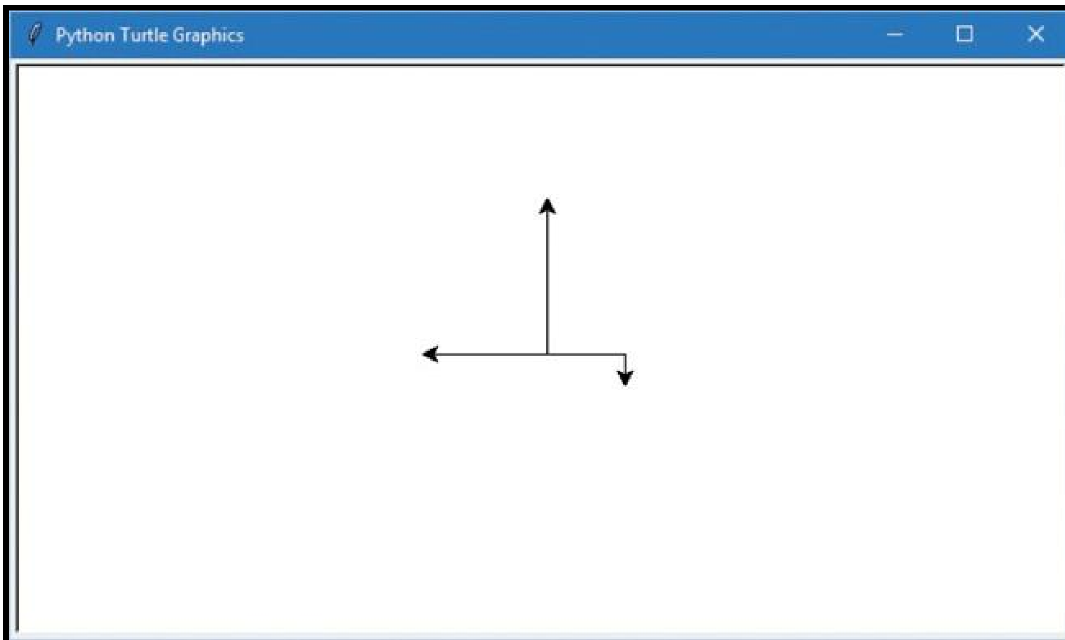


Ora aggiungiamo una terza tartaruga, Jacob, e spostiamola, senza andare a

intralciare Kate o Avery.

```
>>> jacob = turtle.Pen()
>>> jacob.left(180)
>>> jacob.forward(80)
```

Abbiamo creato un nuovo oggetto `Pen` che si chiama `jacob`, poi lo abbiamo fatto ruotare di 180 gradi e spostare in avanti di 80 pixel. Il nostro disegno ora avrà questo aspetto, con tre tartarughe:

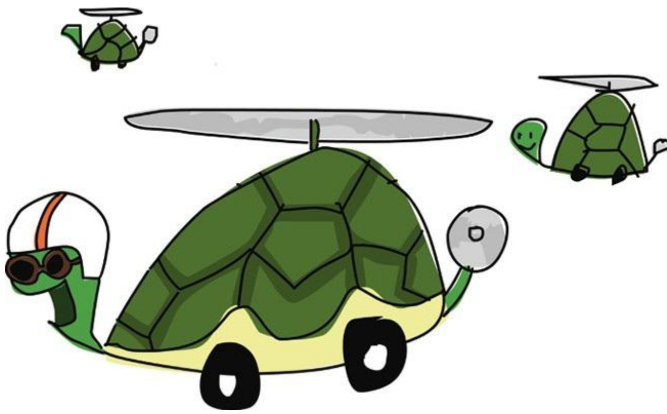


Ricordate che, ogni volta che si chiama `turtle.Pen()` per creare una tartaruga, si aggiunge un nuovo oggetto, indipendente dagli altri. Ciascun oggetto è ancora un'istanza della classe `Pen`, e si possono usare le stesse funzioni su ciascun oggetto, ma, poiché si stanno usando degli oggetti, ciascuna tartaruga si può spostare indipendentemente dalle altre. Come i nostri oggetti giraffa indipendenti (Reginald e Harold), Avery, Kate e Jacob sono tre oggetti tartaruga indipendenti. Se creiamo un nuovo oggetto con lo stesso nome di variabile di un oggetto già creato, il vecchio non necessariamente scomparirà. Provateci: create un'altra tartaruga Kate e provate a spostarla sullo schermo.

ALTRE CARATTERISTICHE UTILI DI OGGETTI E CLASSI

Classi e oggetti facilitano il raggruppamento di funzioni; sono anche estremamente utili quando si vuole ragionare su un programma per parti più piccole.

Per esempio, pensate a un'applicazione davvero molto grande, come un elaboratore di testi o un gioco in 3D. Per la maggior parte delle persone è quasi impossibile capire programmi di dimensioni simili nel loro complesso, perché sono costituiti da così tanto codice. Ma suddividete questi programmi mostruosi in parti più piccole, e ciascun pezzo comincerà ad avere un senso – purché conosciate il linguaggio, ovviamente!



Quando si scrive un programma di grandi dimensioni, suddividerlo in parti consente anche di dividere il lavoro fra più programmatori. I programmi più complicati che utilizzate (come il vostro browser web) sono stati scritti da molte persone, da squadre di persone che lavoravano contemporaneamente su parti diverse, magari in luoghi diversi in giro per il mondo.

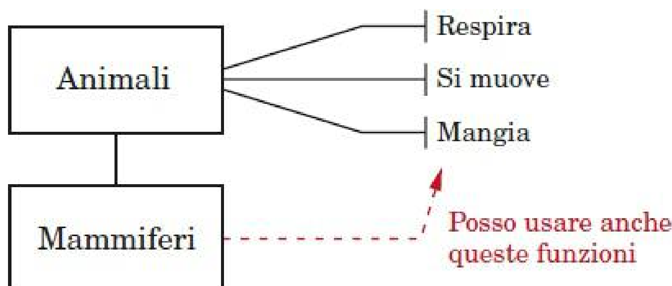
Ora immaginate di voler espandere qualcuna delle classi create in questo capitolo (`Animali`, `Mammiferi` e `Giraffe`): avete troppo lavoro da fare, e volete che i vostri amici vi diano una mano. Potreste dividere il lavoro di scrittura del codice in modo che una persona lavori sulla classe `Animali`, un'altra sulla classe `Mammiferi` e un'altra ancora sulla classe `Giraffe`.

FUNZIONI EREDITATE

Se avete fatto attenzione, vi sarete resi conto che chiunque finisca per lavorare sulla classe `Giraffe` è fortunato, perché quella classe potrà utilizzare anche tutte le funzioni create da chi lavora sulle classi `Animali` e `Mammiferi`. Si dice che la classe `Giraffe` *eredita* le funzioni dalla classe `Mammiferi` che, a sua volta, eredita dalla classe `Animali`. In altre parole, quando creiamo un oggetto giraffa, possiamo usare funzioni definite nella classe `Giraffe`, ma anche quelle definite nelle classi `Mammiferi` e `Animali`.

Analogamente, se creiamo un oggetto mammifero, possiamo usare le funzioni definite nella classe `Mammiferi` e tutte quelle della classe genitrice `Animali`.

Date un'occhiata di nuovo alla relazione fra le classi `Animali`, `Mammiferi` e `Giraffe`. La classe `Animali` è la genitrice di `Mammiferi`, e questa è la genitrice di `Giraffe`.



Anche se Reginald è un oggetto della classe `Giraffe`, possiamo comunque chiamare la funzione `si_muove` definita nella classe `Animali`, perché tutte le funzioni definite in una classe genitrice sono disponibili alle sue classi figlie:

```
>>> reginald = Giraffe()
>>> reginald.si_muove()
si muove
```

In effetti, tutte le funzioni definite nelle classi `Animali` e `Mammiferi` possono essere chiamate dal nostro oggetto `reginald`, perché vengono ereditate:

```
>>> reginald = Giraffe()
>>> reginald.respira()
respira
>>> reginald.mangia()
mangia
>>> reginald.allatta()
allatta
```

FUNZIONI CHE CHIAMANO ALTRE FUNZIONI

Quando chiamiamo funzioni su un oggetto, usiamo il nome di variabile di quell'oggetto. Per esempio, ecco come chiamiamo la funzione `si_muove` sull'oggetto giraffa Reginald:

```
>>> reginald.si_muove()
```

Perché una funzione della classe `Giraffe` chiami la funzione `si_muove`, bisogna usare invece il parametro `self`. Questo parametro è un modo per far sì che una funzione della classe chiami un'altra funzione. Per esempio, aggiungiamo alla classe `Giraffe` una nuova funzione, `cerca_cibo`:

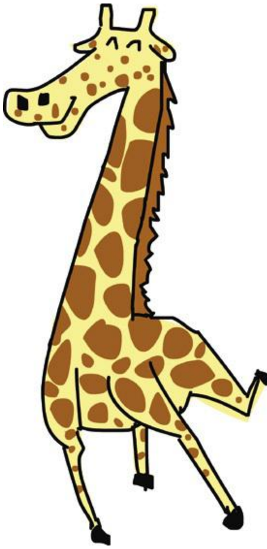
```
>>> class Giraffe(Mammiferi):
        def cerca_cibo(self):
                self.si_muove()
                print('Ho trovato del cibo!')
                self.mangia
```

Ora abbiamo creato una funzione che combina altre due funzioni, cosa molto comune nella programmazione. Spesso, vi capiterà di scrivere una funzione che fa qualcosa di utile, che poi potete utilizzare all'interno di un'altra funzione. (Lo faremo anche noi nel Capitolo 13, dove scriveremo funzioni più complesse per creare un gioco.)

Usiamo `self` per aggiungere qualche altra funzione alla classe `Giraffe`:

```
>>> class Giraffe(Mammiferi):
        def cerca_cibo(self):
                self.si_muove()
                print('Ho trovato del cibo!')
        def mangia_foglie_degli_alberi(self):
                self.mangia()
        def balla(self):
                self_si_muove()
                self_si_muove()
                self_si_muove()
                self_si_muove()
```

Usiamo le funzioni `mangia` e `si_muove` della classe genitrice `Animali` (funzioni ereditate) per definire le funzioni `mangia_foglie_degli_alberi` e `balla`, per la classe `Giraffe`. Aggiungendo funzioni che chiamano altre funzioni in questo modo, quando si creano oggetti di quelle classi, si può chiamare una singola funzione che faccia più di una cosa. Potete vedere qui sotto che cosa succede quando si chiama la funzione `balla`: la nostra giraffa si muove quattro volte (cioè, il testo "si muove" viene stampato quattro volte):



```
>>> reginald = Giraffe()
>>> reginald.balla()
si muove
si muove
si muove
si muove
```

INIZIALIZZARE UN OGGETTO

A volte, quando si crea un oggetto, si vogliono impostare anche dei valori (detti anche *proprietà*) che si utilizzeranno in futuro. Quando si *inizializza* un oggetto, lo si prepara all'uso.

Per esempio, supponiamo di voler indicare quante macchie ci sono sui nostri oggetti giraffa quando vengono creati – cioè quando vengono inizializzati. Per farlo, creiamo una funzione `__init__` (notate che ci sono due caratteri di sottolineatura da ciascuna parte, per un totale di quattro). Questo è un tipo speciale di funzione nelle classi Python e deve avere questo nome. La funzione `init` costituisce un modo per fissare le proprietà di un oggetto quando questo viene creato, e Python chiamerà automaticamente questa funzione quando si crea un nuovo oggetto. Ecco come si usa:

```
>>> class Giraffe:
        def __init__(self, macchie):
            self.macchie_giraffa = macchie
```

Qui come prima cosa abbiamo definito la funzione `init` con due parametri, `self` e `macchie`, con il codice `def __init__ (self, macchie):`. Come le altre funzioni che abbiamo definito nella classe, anche la funzione `init` deve avere `self` come primo parametro. Poi abbiamo impostato il parametro `macchie` a una variabile di oggetto (la sua proprietà) chiamata `macchie_giraffa`, utilizzando il parametro `self`, con il codice `self.macchie_giraffa = macchie`. Potete immaginare che questa riga di codice dica "Prendi il valore del parametro `macchie` e salvalo per più tardi (utilizzando la variabile di oggetto `macchie_giraffa`)". Come una funzione in una classe può chiamare un'altra funzione utilizzando il parametro `self`, anche alle variabili della classe si può accedere con `self`.

Poi, se creiamo un paio di nuovi oggetti giraffa (Ozwald e Gertrude) e stampiamo il numero delle loro macchie, possiamo vedere la funzione di inizializzazione all'opera:

```
>>> ozwald = Giraffe(100)
>>> gertrude = Giraffe(150)
>>> print(ozwald.macchie_giraffa)
100
>>> print(gertrude.macchie_giraffa)
150
```

Come prima cosa, abbiamo creato un'istanza della classe `Giraffe`, utilizzando il valore di parametro `100`. Questo fa sì che venga chiamata la funzione `__init__` con il valore 100 per il parametro `macchie`. Poi abbiamo creato un'altra istanza della classe `Giraffe`, questa volta con il valore `150`.
Infine, abbiamo stampato la variabile oggetto `macchie_giraffa` per ciascuno dei nostri oggetti giraffa, e possiamo vedere che i risultati sono 100 e 150. Ha funzionato!

Ricordate, quando si crea un oggetto di una classe, come ozwald qui sopra, si può far riferimento alle sue variabili o alle sue funzioni utilizzando l'operatore punto e il nome della variabile o della funzione che si vuole usare (per esempio, `ozwald.macchie_giraffa`). Quando invece si creano funzioni all'interno di una classe, si fa riferimento a quelle stesse variabili (e ad altre funzioni) utilizzando il parametro `self` (`self.macchie_giraffa`).

CHE COSA AVETE IMPARATO

In questo capitolo, abbiamo usato le classi per creare categorie di cose e abbiamo creato oggetti (istanze) di quelle classi. Avete visto come una classe figlia erediti le funzioni della sua genitrice e che, anche se due oggetti appartengono alla stessa classe, non sono necessariamente dei cloni. Per esempio, un oggetto giraffa può avere il suo numero di macchie. Avete imparato come chiamare (eseguire) funzioni su un oggetto e come le variabili oggetto siano un modo per salvare dei valori in quegli oggetti. Infine, abbiamo usato il parametro `self` nelle funzioni per far riferimento ad altre funzioni e variabili. Questi sono concetti fondamentali di Python, e li incontrerete ancora molte volte continuando a leggere questo libro.

ROMPICAPO DI PROGRAMMAZIONE

Alcune delle idee viste in questo capitolo acquisteranno maggiore significato quanto più le userete. Provate a svolgere gli esempi seguenti, poi cercate le risposte all'indirizzo *http://python-for-kids.com/*.

1. LA DANZA DELLA GIRAFFA

Aggiungete alla classe `Giraffe` delle funzioni per far spostare avanti e indietro lo zoccolo sinistro e destro della giraffa. Per esempio, una funzione per far spostare avanti la zampa sinistra potrebbe essere scritta in questo modo:

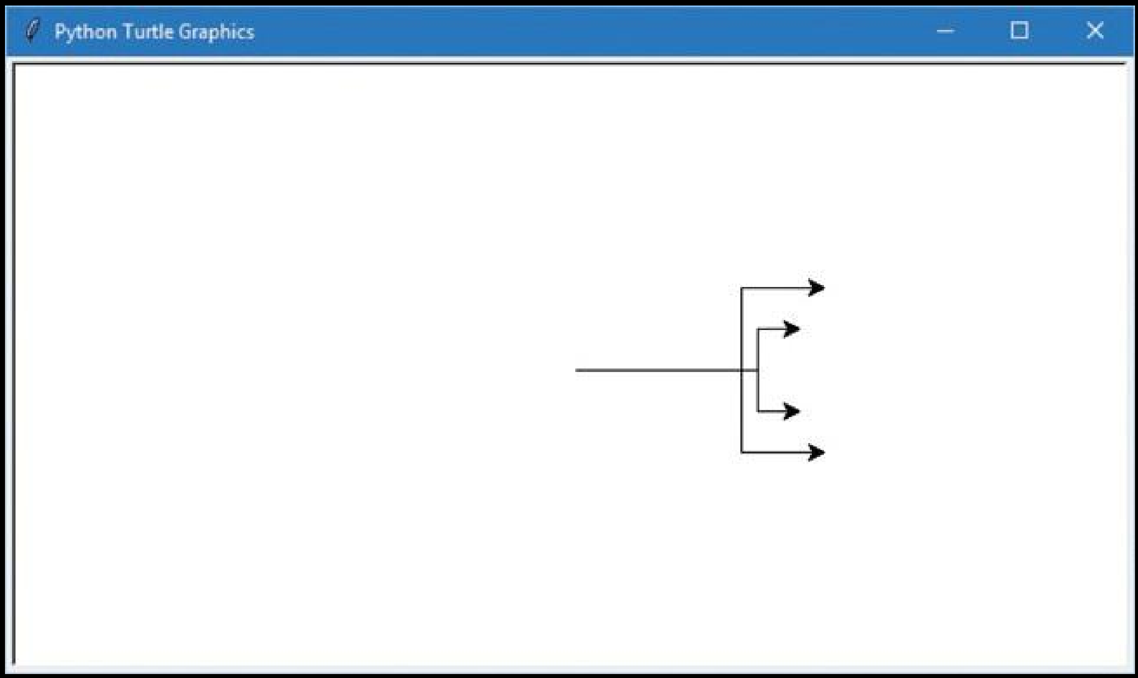
```
>>> def zampa_sinistra_avanti(self):
        print('zampa sinistra avanti')
```

Poi create una funzione `danza` per insegnare a Reginald come si balla (la funzione chiamerà le quattro funzioni appena create per le zampe). Il risultato della chiamata di questa nuova funzione sarà una danza molto semplice:

```
>>> reginald = Giraffe()
>>> reginald.danza
zampa sinistra avanti
zampa sinistra indietro
zampa destra avanti
zampa destra indietro
zampa sinistra indietro
zampa destra indietro
zampa destra avanti
zampa sinistra avanti
```

2. FORCONE PER TARTARUGHE

Create questo disegno di un forcone utilizzando quattro oggetti tartaruga `Pen` (la lunghezza esatta dei singoli segmenti non è importante). Ricordate, come prima cosa, di importare il modulo `turtle`!


Python Turtle Graphics

9
LE FUNZIONI INTERNE DI PYTHON

Python mette a disposizione una cassetta degli attrezzi per la programmazione ben fornita, in cui si trovano, fra le altre cose, molte funzioni e molti moduli già pronti all'uso. Come un martello fidato o una chiave inglese, questi strumenti incorporati (pezzi di codice, in realtà) possono rendere molto più facile scrivere programmi.

Come abbiamo visto nel Capitolo 7, i moduli devono essere importati, per poter essere utilizzati, mentre non c'è bisogno di importare le funzioni *interne*; sono disponibili appena la shell di Python si avvia.

In questo capitolo, esamineremo alcune delle funzioni interne più utili, poi ci concentreremo su una in particolare: la funzione `open`, che consente di aprire file per poterne leggere i contenuti e per poter scrivere al loro interno.

COME SI USANO LE FUNZIONI INTERNE

Considereremo qui dodici funzioni interne, usate spesso dai programmatori Python. Vedremo che cosa fanno e come si usano, e faremo qualche esempio di come possono aiutare nella scrittura di programmi.

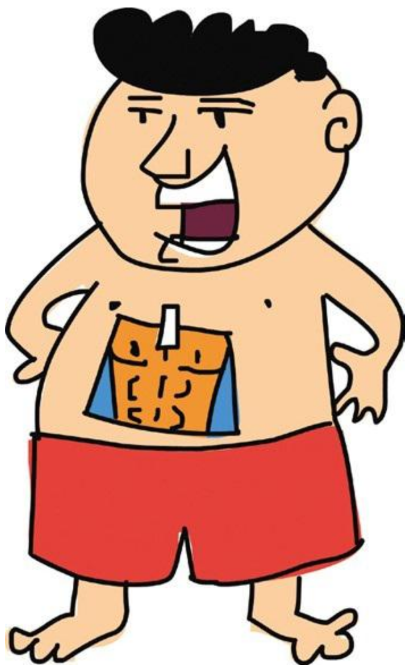
LA FUNZIONE ABS

La funzione abs restituisce il *valore assoluto* di un numero, che è il valore di quel numero senza segno. Per esempio, il valore assoluto di 10 è 10, il valore assoluto di -10 è 10.

Per usare la funzione `abs`, basta chiamarla con un numero o una variabile come parametro:

```
>>> print(abs(10))
10
>>> print(abs(-10))
10
```

Potete usare la funzione `abs` per esempio per calcolare la quantità assoluta di movimento di un personaggio in un gioco, indipendentemente dalla direzione in cui si è spostato. Per esempio, supponiamo che il personaggio faccia tre passi a destra (3 positivo) e poi dieci passi a sinistra (10 negativo, ovvero -10). Se non ci interessa la direzione (positiva o negativa), il valore assoluto di questi numeri sarebbe 3 e 10, Potreste usare questi valori in un gioco da tavolo in cui lanciate due dadi e poi fate spostare il personaggio di un numero massimo di passi in qualsiasi direzione, sulla base del punteggio totale dei due dadi. Ora, se memorizzate quel numero di passi in una variabile, potete determinare se il personaggio si muove con il codice che segue. Vogliamo visualizzare qualche informazione quando il giocatore ha deciso di spostarsi (qui, visualizzeremo solo “Il personaggio si muove”):



```
>>> steps = -3
>>> if abs(steps) > 0:
        print('Il personaggio si muove')
```

Se non avessimo usato `abs`, l'enunciato `if` avrebbe dovuto essere di questo tipo:

```
>>> steps = -3
>>> if steps < 0 or steps > 0:
        print('Il personaggio si muove')
```

Come potete vedere, l'uso di `abs` rende l'enunciato `if` un po' più breve e più facile da capire.

LA FUNZIONE BOOL

Il nome `bool` è abbreviazione di *booleano*, l'aggettivo che i programmatori usano per descrive un tipo di dati che può avere solo due valori, "vero" (*True*) o "falso" (*False*).

La funzione `bool` prende un unico parametro e restituisce o `True` o `False` a seconda del suo valore. Se si usa `bool` per i numeri, 0 restituisce `False`, mentre ogni altro numero restituisce `True`. Ecco qualche esempio:

```
>>> print(bool(0))
False
>>> print(bool(1))
True
>>> print(bool(1123.23))
True
>>> print(bool(-500))
True
```

Se si usa `bool` con altri valori, per esempio stringhe, restituisce `False` se la stringa non ha alcun valore (ovvero la parola chiave `None`, è una stringa vuota). In ogni altro

caso restituisce `True`:

```
>>> print(bool(None))
False
>>> print(bool('a'))
True
>>> print(bool(' '))
True
>>> print(bool('Come definiresti un maiale che fa karate?'))
True
```

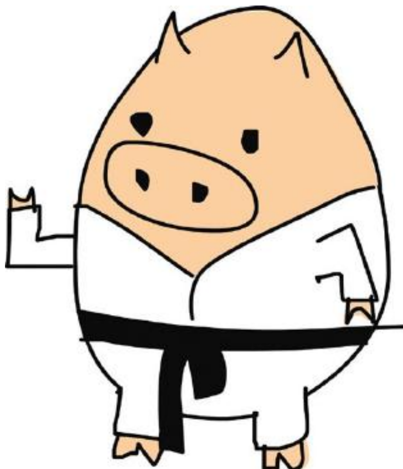
La funzione `bool` restituirà `False` anche per liste, tuple e mappe che non contengono alcun valore, `True` se invece contengono qualche valore:

```
>>> mia_lista_stupida = []
>>> print(bool(mia_lista_stupida))
False
>>> mia_lista_stupida = ['s', 't', 'u', 'p', 'i', 'd', 'a']
>>> print(bool(mia_lista_stupida))
True
```

Potete usare `bool` quando dovete decidere se un valore sia stato impostato o meno. Per esempio, se chiediamo a chi usa il nostro programma di inserire l'anno in cui è nato, un `if` potrebbe usare `bool` per verificare il valore inserito:

```
>>> anno = input('Anno di nascita: ')
Anno di nascita:
>>> if not bool(anno.rstrip()):
        print("Devi inserire un valore per l'anno di nascita")
Devi inserire un valore per l'anno di nascita
```

La prima riga usa `input` per immagazzinare nella variabile `anno` ciò che l'utente inserisce dalla tastiera. Premendo INVIO sulla riga successiva (senza scrivere nulla), nella variabile viene memorizzato il valore di INVIO. (Nel Capitolo 7 abbiamo usato `sys.stdin.readline()`, che è un altro modo per fare la stessa cosa.)



Sulla riga successiva, l'enunciato `if` controlla il valore booleano della variabile, dopo aver utilizzato la funzione `rstrip` (che elimina i caratteri spazio e INVIO alla fine

della stringa). Dato che l'utente qui non ha inserito nulla, `bool` restituisce `False`. Poiché l'enunciato `if` usa la parola chiave `not`, il suo significato è "fai questo se la funzione non restituisce True", perciò il codice stampa `Devi inserire un valore per l'anno di nascita` sulla riga successiva.

LA FUNZIONE DIR

La funzione `dir` (abbreviazione di *directory*) restituisce informazioni su qualsiasi valore. Fondamentalmente, vi dice quali funzioni possono essere usate con quel valore, elencandole in ordine alfabetico.

Per esempio, per vedere quali funzioni siano disponibili per un valore lista, potete scrivere:

```
>>> dir(['una', 'breve', 'lista'])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__dir__', '__doc__', '__eq__', '__format__',
'__ge__', '__getattribute__', '__getitem__', '__gt__',
'__hash__', '__iadd__', '__imul__', '__init__', '__iter__',
'__le__', '__len__', '__lt__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__reversed__',
'__rmul__', '__setattr__', '__setitem__', '__sizeof__', '__
str__', '__subclasshook__', 'append', 'clear', 'copy', 'count',
'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

La funzione `dir` è utilizzabile su quasi tutto, in particolare per stringhe, numeri, funzioni, moduli, oggetti e classi. Qualche volta le informazioni che restituisce, però, possono essere poco utili. Per esempio, se chiamate `dir` per il numero 1, visualizza una serie di funzioni speciali (quelle che iniziano e terminano con gli underscore, i trattini di sottolineatura) usate da Python stesso, il che non è molto utile (di solito si può ignorare la maggior parte di queste funzioni):

```
>>> dir(1)
['__abs__', '__add__', '__and__', '__bool__', '__ceil__',
'__class__', '__delattr__', '__dir__', '__divmod__', '__doc__',
'__eq__', '__float__', '__floor__', '__floordiv__', '__for-
mat__', '__ge__', '__getattribute__', '__getnewargs__', '__gt__',
'__hash__', '__index__', '__init__', '__int__', '__invert__',
'__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__',
'__neg__', '__new__', '__or__', '__pos__', '__pow__', '__radd__',
'__rand__', '__rdivmod__', '__reduce__', '__reduce_ex__',
'__repr__', '__rfloordiv__', '__rlshift__', '__rmod__',
'__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__',
'__rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__
setattr__', '__sizeof__', '__str__', '__sub__',
'__subclasshook__', '__truediv__', '__trunc__', '__xor__',
'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag',
'numerator', 'real', 'to_bytes']
```

La funzione `dir` può essere utile quando si ha una variabile e si vuole capire rapidamente che cosa si può fare con quella variabile. Per esempio, eseguite `dir` con la variabile `popcorn` che contiene un valore stringa, e otterrete la lista delle funzioni messe a disposizione dalla classe `string` (tutte le stringhe appartengono alla classe `string`):

```
>>> popcorn = 'Amo il popcorn!'
>>> dir(popcorn)
['__add__', '__class__', '__contains__', '__delattr__',
'__dir__', '__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__getnewargs__', '__gt__',
'__hash__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mod__', '__mul__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__rmod__',
'__rmul__', '__setattr__', '__sizeof__', '__str__',
'__subclasshook__', 'capitalize', 'casefold', 'center', 'count',
'encode', 'endswith', 'expandtabs', 'find', 'format',
'format_map', 'index', 'isalnum', 'isalpha', 'isdecimal',
'isdigit', 'isidentifier', 'islower', 'isnumeric', 'isprint-
able', 'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'maketrans', 'partition', 'replace', 'rfind', 'rindex',
'rjust', 'rpartition', 'rsplit', 'rstrip', 'split', 'splitlines',
'startswith', 'strip', 'swapcase', 'title', 'translate', 'upper',
'zfill']
```

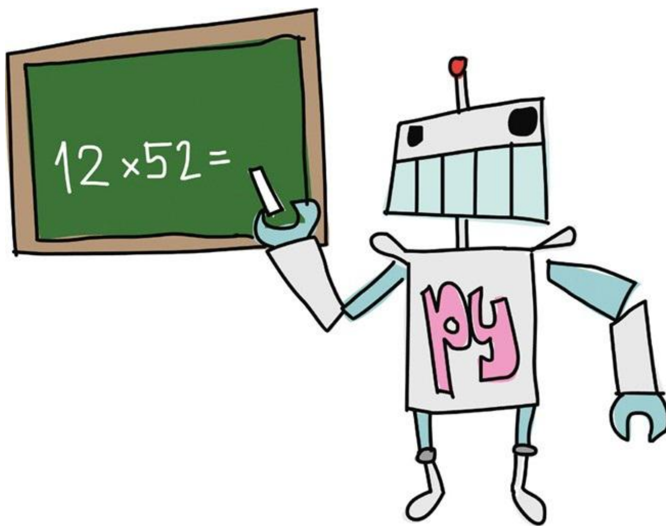
A questo punto, potete usare `help` per vedere una breve descrizione di qualsiasi funzione nella lista. Ecco un esempio:

```
>>> help(popcorn.upper)
Help on built-in function upper:

upper(...) method of builtins.str instance
    S.upper() -> str
    Return a copy of S converted to uppercase.
```

Le informazioni restituite possono risultare poco chiare, perciò diamo un'occhiata più da vicino. I tre puntini (…) significano che `upper` è una funzione interna della classe `string` e, in questo caso, che non assume parametri (è il succo delle parole che seguono la parentesi con i tre puntini). La freccia `(->)` sulla riga successiva significa che la funzione restituisce una stringa `(str)`. L'ultima riga dà una breve descrizione di quello che fa la funzione: restituisce una copia di `S` (la stringa) convertita in tutte maiuscole (*uppercase*).

LA FUNZIONE EVAL

La funzione `eval` (abbreviazione di *evalutation*, valutazione) prende come parametro una stringa, e la elabora come se fosse un'espressione Python. Per esempio, `eval('print("wow")')` esegue l'enunciato `print("wow")`. La funzione `eval` opera solo con espressioni semplici, come questa:




```
>>> eval('10*5')
50
```

Espressioni disposte su più righe (come gli enunciati `if`) in genere non verranno valutate, come in questo esempio:

```
>>> eval('''if True:
print("questo non funziona")''')
Traceback (most recent call last):
  File "<pyshell#48>", line 2, in <module>
    print("questo non funziona")''')
  File "<string>", line 1
    if True:
     ^
SyntaxError: invalid syntax
```

La funzione `eval` si usa spesso per convertire l'input dell'utente in espressioni Python. Per esempio, si potrebbe scrivere un semplice programma "calcolatrice" che legga espressioni matematiche e poi ne calcoli (valuti) la soluzione.

Dato che l'input dell'utente è letto come stringa, Python deve convertirlo in numeri e operatori, prima di eseguire qualsiasi calcolo. La funzione `eval` rende facile la conversione:

```
>>> tuo_calcolo = input('Inserisci un calcolo: ')
Inserisci un calcolo: 12*52
>>> eval(tuo_calcolo)
624
```

Nell'esempio, usiamo `input` per leggere quanto inserito dall'utente nella variabile `tuo_calcolo`. Nella riga successiva, inseriamo l'espressione `32*52`, poi usiamo `eval` per eseguire il calcolo; il risultato viene stampato sulla riga seguente.

LA FUNZIONE EXEC

La funzione `exec` è come `eval`, tranne che la si può usare per eseguire programmi più complessi. La differenza è che `eval` restituisce un valore (qualcosa che si può salvare

in una variabile), mentre `exec` no. Ecco un esempio:

```
>>> programmino = '''print('panino')
print('al prosciutto')'''
>>> exec(programmino)
panino
al prosciutto
```

Nelle prime due righe, creiamo una variabile con una stringa multilinea che contiene due enunciati `print`, poi usiamo `exec` per eseguire la stringa.

Si potrebbe usare `exec` per eseguire piccoli programmi che un programma Python legge da file: programmi dentro programmi! Può essere molto utile, quando si scrivono applicazioni lunghe e complesse. Per esempio, si potrebbe creare un gioco Dueling Robots in cui due robot si spostano sullo schermo e cercano di attaccarsi a vicenda. I giocatori darebbero istruzioni ai loro robot sotto forma di miniprogrammi Python e il gioco leggerebbe le istruzioni e userebbe `exec` per eseguirle.

LA FUNZIONE FLOAT

La funzione `float` converte una stringa o un numero in un numero in virgola mobile (*floating point*), cioè un numero con posizioni decimali (che è anche un numero reale). Per esempio, il numero 10 è un *intero*, mentre 10.0, 10.1 e 10.253 (in Python si usa, come nel sistema anglosassone, il punto al posto della virgola) sono tutti numeri in virgola mobile. Si possono usare i numeri in virgola mobile (anziché gli interi) per esempio quando si scrive un programma che comporti calcoli di somme di denaro, oppure nei programmi di grafica (giochi 3D, per esempio) per calcolare come e dove disegnare qualcosa sullo schermo.

Si può convertire una stringa in un numero in virgola mobile semplicemente chiamando `float`:

```
>>> float('12')
12.0
```

Si può usare anche il punto decimale nelle stringhe:

```
>>> float('123.456789')
123.456789
```

Si può usare `float` per convertire in numeri i valori inseriti nel programma, cosa particolarmente utile quando si deve confrontare il valore inserito dall'utente con qualche altro valore. Per esempio, per verificare se l'età di una persona è superiore a un certo numero, potremmo fare così:

```
>>> tua_età = input('Inserisci la tua età: ')
Inserisci la tua età: 20
>>> età = float(tua_età)
>>> if età > 13:
        print('Hai %s anni di troppo' % (età - 13))
Hai 7.0 anni di troppo
```

LA FUNZIONE INT

La funzione `int` converte una stringa o un numero qualsiasi in un intero, il che significa sostanzialmente che tutto ciò che eventualmente compariva dopo il punto decimale viene eliminato. Per esempio, ecco come convertire un numero in virgola mobile in un semplice intero:

```
>>> int(123.456)
123
```

Questo esempio converte una stringa in un intero:

```
>>> int('123')
123
```

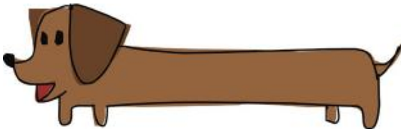
Provate però a convertire in un intero una stringa contenente un numero in virgola mobile e otterrete un messaggio d'errore. Per esempio:

```
>>> int('123.456')
Traceback (most recent call last):
  File "<pyshell#71>", line 1, in <module>
    int('123.456')
ValueError: invalid literal for int() with base 10: '123.456'
```

Come potete vedere, il risultato è un messaggio ValueError.

LA FUNZIONE LEN

La funzione `len` restituisce la lunghezza di un oggetto o, nel caso di una stringa, il numero dei caratteri che la compongono. Per esempio, per avere la lunghezza di `questa è una stringa di prova`, si può fare così:



```
>>> len('questa è una stringa di prova')
29
```

Se usata con una lista o una tupla, `len` restituisce il numero degli elementi che la compongono:

```
>>> lista_creature = ['unicorni', 'ciclopi', 'fate', 'elfi',
'draghi', 'troll']
>>> print(len(lista_creature))
6
```

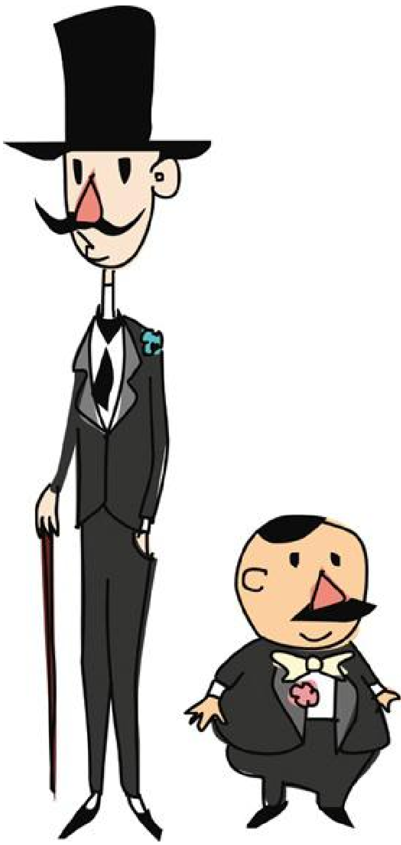
Anche quando è usata con una mappa, `len` restituisce il numero dei suoi elementi:

```
>>> mappa_nemici = {'Batman' : 'Joker',
               'Superman' : 'Lex Luthor',
               'Spiderman' : 'Green Goblin'}
>>> print(len(mappa_nemici))
3
```

La funzione `len` è particolarmente utile quando si lavora con i cicli. Per esempio, la si può usare per visualizzare l'indice della posizione degli elementi in una lista:

```
  >>> frutta = ['mela', 'banana', 'mandarino', 'frutto del drago']
❶ >>> lunghezza = len(frutta)
❷ >>> for x in range(0, lunghezza):
❸         print('il frutto nella posizione %s è %s' % (x,
frutta[x]))

il frutto nella posizione 0 è mela
il frutto nella posizione 1 è banana
il frutto nella posizione 2 è mandarino
il frutto nella posizione 3 è frutto del drago
```

Qui, memorizziamo la lunghezza della lista nella variabile `lunghezza` in ❶, poi usiamo quella variabile nella funzione `range` per creare il ciclo in ❷. In ❸, mentre si procede ciclicamente lungo tutti gli elementi della lista, stampiamo un messaggio che indica la posizione e il valore dell'elemento considerato. Si potrebbe usare la funzione `len` anche, per esempio, se si avesse una lista di stringhe e si volesse stampare un elemento della stringa ogni due o ogni tre.



LE FUNZIONI MAX E MIN

La funzione `max` restituisce l'elemento più grande in una lista, una tupla o una stringa. Per esempio, eccola usata con una lista di numeri:

```
>>> numeri = [5, 4, 10, 30, 22]
>>> print(max(numeri))
30
```

La si può usare anche con una stringa separata da virgole o spazi:

```
>>> stringa = 's,t,r,i,n,g,a,S,T,R,I,N,G,A'
>>> print(max(stringa))
t
```

Come si vede in questo caso, le lettere vengono ordinate alfabeticamente, e le minuscole vengono dopo le maiuscole, perciò *t* è "più grande" di *T*.

Non è necessario comunque usare liste, tuple o stringhe. Si può usare `max` anche direttamente, e inserire gli elementi che si vogliono confrontare fra parentesi, come parametri:

```
>>> print(max(10, 300, 450, 50, 90))
450
```

La funzione `min` si comporta nello stesso modo, tranne che restituisce l'elemento più piccolo della lista, tupla o stringa. Eccola applicata alla nostra lista di numeri:

```
>>> numeri = [5, 4, 10, 30, 22]
>>> print(min(numeri))
4
```

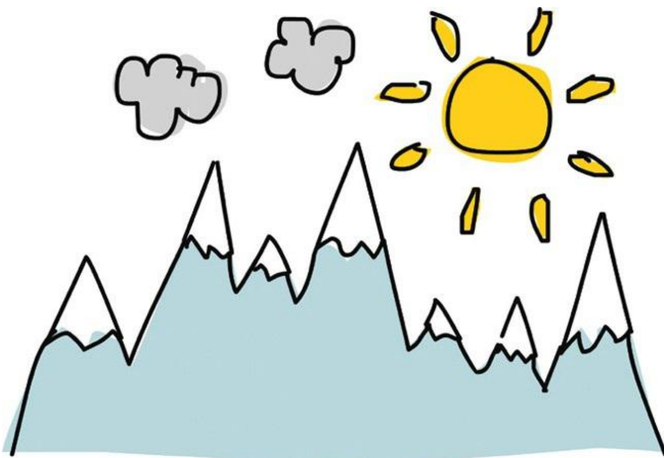
Supponiamo di avere un gioco, con quattro giocatori, in cui ciascuno deve tirare a indovinare e indicare un numero minore del vostro. Se qualche giocatore propone un numero maggiore del vostro, tutti perdono, ma se tutti propongono numeri più bassi, vincono. Potremmo usare `max` per scoprire rapidamente se tutti i numeri che gli altri giocatori hanno proposto sono più bassi:

```
>>> indovina_il_numero = 61
>>> risposte_giocatori = [12, 15, 70, 45]
>>> if max(risposte_giocatori) > indovina_il_numero:
        print('Ahi! Avete perso tutti')
else:
        print('Avete vinto')

Ahi! Avete perso tutti
```

In questo esempio, memorizziamo il numero da indovinare usando la variabile `indovina_il_numero`. I numeri proposti dai giocatori sono memorizzati nella lista `risposte_giocatori`. L'enunciato `if` confronta la risposta più alta con il numero in `indovina_il_numero`, e se qualche giocatore ha proposto un numero più alto, viene stampato il messaggio "Ahi! Avete perso tutti".

LA FUNZIONE RANGE

La funzione `range`, come abbiamo già visto, viene usata soprattutto nei cicli `for`, per percorrere una sezione di codice uno specifico numero di volte. I primi due parametri passati a `range` sono chiamati *start* e *stop*. Abbiamo visto usata `range` con questi due parametri nell'esempio precedente, insieme alla funzione `len`, per effettuare un ciclo.



I numeri generati da `range` iniziano da quello dato come primo parametro e finiscono con quello passato come secondo parametro. Per esempio, ecco che cosa succede se stampiamo i numeri creati da `range` fra 0 e 5:

```
>>> for x in range(0, 5):
        print(x)

0
1
2
3
4
```

La funzione `range` in realtà restituisce un oggetto speciale, chiamato *iteratore*, che ripete un'azione per un certo numero di volte. In questo caso, restituisce il numero successivo, ogni volta che viene chiamata.

Si può convertire l'iteratore in una lista (mediante la funzione `list`). Se poi si stampa il valore restituito dalla chiamata a `range`, si vedranno i numeri che contiene:

```
>>> print(list(range(0, 5)))
[0, 1, 2, 3, 4]
```

Si può passare a `range` anche un terzo parametro, chiamato *step* (passo). Se il valore step non è incluso, viene utilizzato come passo 1.

Che cosa succede, invece, se si indica il numero 2 come passo?

```
>>> contare_per_due = list(range(0, 30, 2))
>>> print(contare_per_due)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28]
```

Ogni numero nella lista è più grande di due unità rispetto al precedente, e la lista finisce con il numero 28, che è 30 – 2. Si può indicare anche un passo negativo:

```
>>> conto_alla_rovescia = list(range(40, 10, -2))
>>> print(conto_alla_rovescia)
[40, 38, 36, 34, 32, 30, 28, 26, 24, 22, 20, 18, 16, 14, 12]
```

LA FUNZIONE SUM

La funzione `sum` somma gli elementi di una lista e restituisce il totale. Ecco un esempio:

```
>>> lista_numeri = list(range(0, 500, 50))
>>> print(lista_numeri)
[0, 50, 100, 150, 200, 250, 300, 350, 400, 450]
>>> print(sum(lista_numeri))
2250
```

Sulla prima riga, creiamo una lista di numeri compresi fra 0 e 500, usando la funzione `range` con passo 50. Poi stampiamo la lista per vedere il risultato. Infine, passando la variabile `lista_numeri` alla funzione `sum` con `print(sum(lista_numeri))`, tutti gli elementi della lista vengono sommati e viene stampato il totale, 2250.

LAVORARE CON I FILE

I file di Python sono uguali ad altri file sul vostro computer: documenti, immagini, musica, giochi… in effetti, tutto nel vostro computer è salvato come file.

Vediamo come in Python si aprano i file, con la funzione `open`, per potervi lavorare. Prima però dobbiamo creare un nuovo file con cui fare le nostre prove.

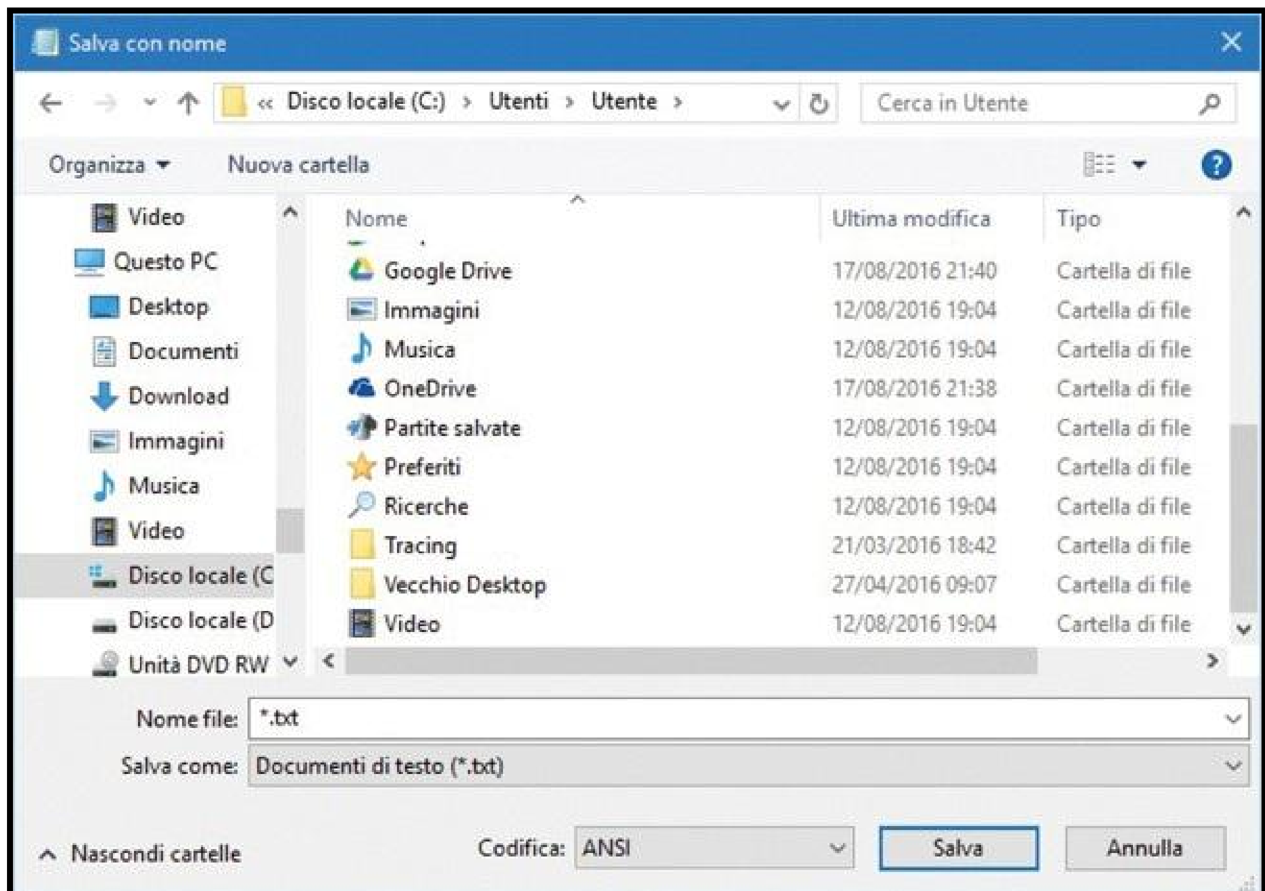
CREARE UN FILE DI PROVA

Faremo qualche esperimento con un file di testo, che chiameremo *test.txt*. Seguite la procedura per il sistema operativo che utilizzate.

CREARE UN NUOVO FILE IN WINDOWS

Se usate Windows, seguite questi passi per creare *test.txt:*

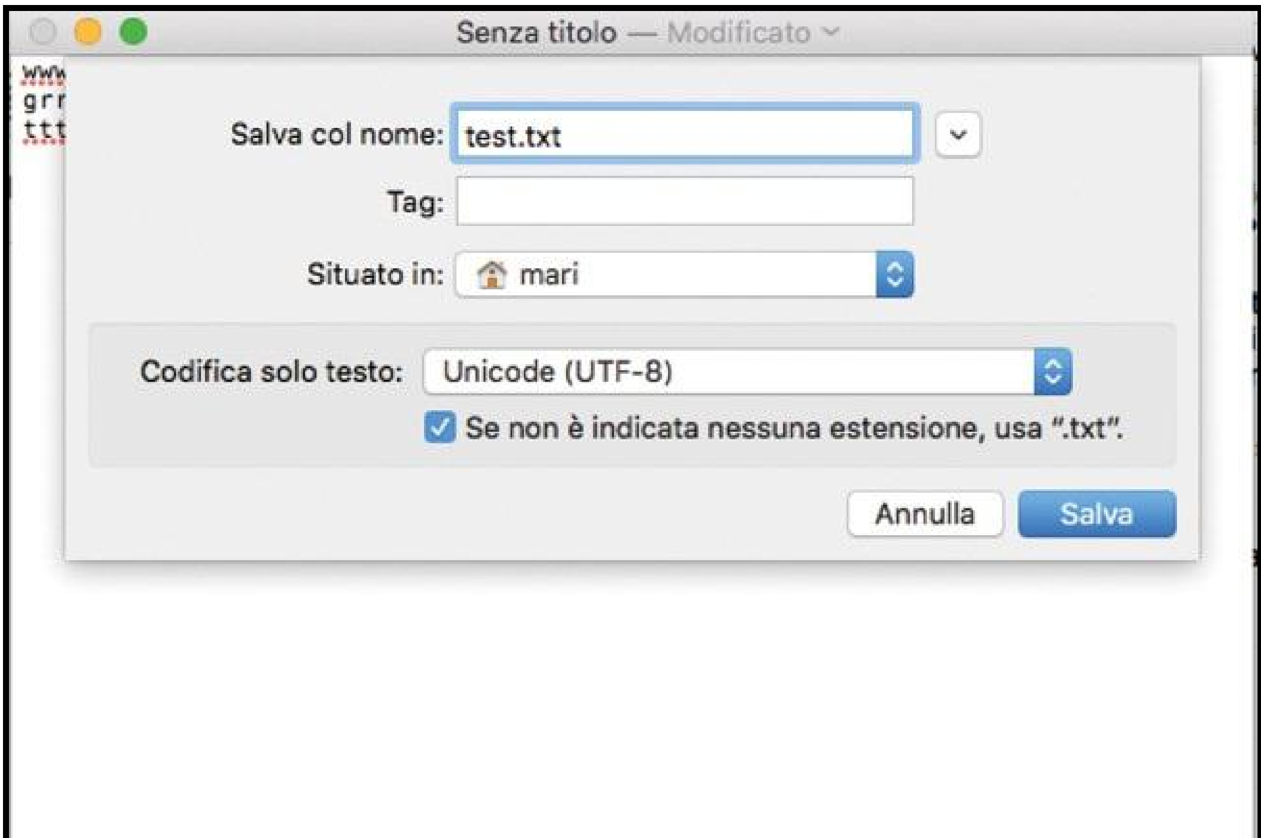
1. Selezionate **Start** > **Tutti i programmi** > **Accessori** > **Blocco note**.
2. Scrivete qualche riga in un file vuoto.
3. Selezionate **File** > **Salva**.
4. Quando compare la finestra di dialogo, selezionate l'unità *C:* facendo un doppio clic su **Computer** e poi su **Disco locale (C:)**.
5. Fate un doppio clic sulla cartella *Utenti* e poi un doppio clic sul vostro nome utente.
6. Scrivete *test.txt* nella casella **Nome file**, in basso nella finestra di dialogo.
7. Infine, fate clic sul pulsante **Salva**.



CREARE UN NUOVO FILE IN MAC OS X

Se usate un Mac, seguite questi passi per creare *test.txt:*

1. Fate clic sull'icona **Spotlight** nella barra dei menu nella parte superiore dello schermo.
2. Scrivete *TextEdit* nella casella di ricerca.
3. Nella sezione Applicazioni comparirà TextEdit. Fate un clic per aprire l'editor (potete trovare TextEdit anche nella cartella Applicazioni del Finder).
4. Scrivete qualche riga di testo nel file vuoto.
5. Selezionate **Formato** > **Solo testo**.
6. Selezionate **File** > **Salva**.
7. Nella casella **Salva con nome**, scrivete *test.txt*.
8. Nell'elenco **Situato in**, fate clic sul vostro nome utente (il nome con cui avete effettuato l'accesso, oppure il nome della persona che possiede il computer che state usando).
9. Infine, fate clic sul pulsante **Salva**.

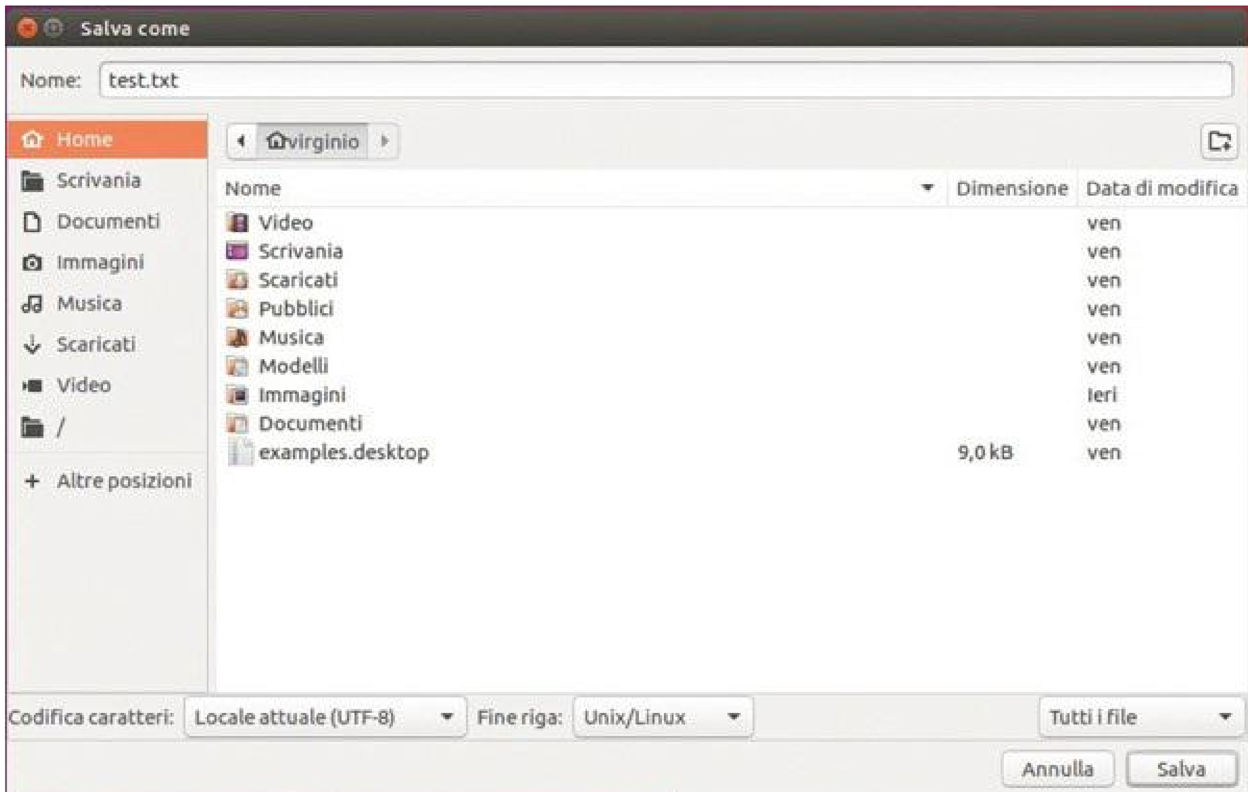


CREARE UN NUOVO FILE IN UBUNTU

Se usate Ubuntu, seguite questi passi per creare *test.txt:*

1. Aprite il vostro editor, che può essere Text Editor o gedit o altro ancora. Nel caso, cercatelo nel Software Center.
2. Inserite qualche riga di testo nell'editor.
3. Selezionate **File** > **Salva** (in gedit il pulsante **Salva**)

4. Nella casella **Nome**, scrivete *test.txt* come nome di file. Come directory di salvataggio può darsi sia già selezionata la vostra directory Home, ma, se non lo è, selezionatela. (Eventualmente, la directory home è il nome utente con cui avete effettuato l'accesso.)
5. Fate clic sul pulsante **Salva**.



APRIRE UN FILE IN PYTHON

La funzione interna `open` apre un file nella shell di Python e ne visualizza i contenuti. Come dire alla funzione quale file aprire dipende dal sistema operativo. Guardate l'esempio per un file Windows, poi leggete i paragrafi specifici per Mac o Ubuntu, se usate uno di questi sistemi.

APRIRE UN FILE IN WINDOWS

Se usate Windows, inserite questo codice per aprire *test.txt:*

```
>>> test_file = open('C:\\Users\\<qui scrivete il vostro nome>\\
test.txt')
>>> text = test_file.read()
>>> print(text)
La vispa Teresa
avea tra l'erbetta
a volo sorpresa
gentil farfalletta
```

Nella prima riga, usiamo `open`, che restituisce un oggetto file con le funzioni per lavorare sui file. Il parametro utilizzato con la funzione `open` è una stringa che dice a Python dove trovare il file. Se usate Windows, avete salvato *test. txt* nella vostra

directory home sull’unità *C:*, perciò dovete specificare la posizione del vostro file con la stringa C:\\Users\\<qui scrivete il vostro nome>\\test.txt.

Le due barre rovesciate nel nome del file Windows dicono a Python che la barra rovesciata è solo quello, non qualche tipo di comando. (Come abbiamo visto nel Capitolo 3, le barre rovesciate da sole hanno un significato speciale in Python, in particolare nelle stringhe.) Salviamo l’oggetto file nella variabile test_file.

APRIRE UN FILE IN MAC OS X

Se usate Mac OS X, dovrete inserire una posizione diversa nella prima riga dell’esempio di Windows, per aprire *test.txt*. Nella stringa, usate il nome utente su cui avete fatto clic quando avete salvato il file di testo. Per esempio, se il nome utente era *martarossi*, il parametro di open dovrà essere scritto in questo modo:

```
>>> test_file = open ('/Users/martarossi/test.txt')
```

APRIRE UN FILE IN UBUNTU

Se usate Ubuntu, dovrete inserire una posizione diversa nella prima riga dell’esempio di Windows, per aprire *test.txt*. Nella stringa, usate il nome utente su cui avete fatto clic quando avete salvato il file di testo. Per esempio, se il nome utente era *giacomo*, il parametro di open dovrà essere scritto in questo modo:

```
>>> test_file = open ('/home/giacomo/test.txt')
```

SCRIVERE NEI FILE

L’oggetto file restituito da open ha altre funzioni, oltre a read. Possiamo creare un nuovo file vuoto usando un secondo parametro, la stringa 'w', quando chiamiamo la funzione:

```
>>> test_file = open('C:\\miofile.txt', 'w')
```

Il parametro 'w' dice a Python che vogliamo scrivere nell’oggetto file, anziché leggere i suoi contenuti.

Adesso possiamo aggiungere delle informazioni in questo nuovo file, mediante la funzione write (“scrivi”):

```
>>> test_file = open('C:\\ miofile.txt', 'w')
>>> test_file.write('Questo è il mio file di prova')
29
```

Il numero che compare nella riga successiva è il numero dei caratteri che sono stati scritti nel file.

Infine, dobbiamo dire a Python che abbiamo finito di scrivere nel file, utilizzando la funzione close:

```
>>> test_file = open('C:\\ miofile.txt', 'w')
>>> test_file.write('Che cosa è verde e rumoroso? Una rana
urlatrice! ')
>>> test_file.close()
```

Ora, se aprite il file con il vostro text editor, vedrete che contiene il testo “Che cosa è grande e rumoroso? Una rana urlatrice!”. Oppure, potete usare Python per leggerlo:

```
>>> test_file = open('C:\\ miofile.txt')
>>> print(test_file.read())
Che cosa è verde e rumoroso? Una rana
urlatrice!
```

CHE COSA AVETE IMPARATO

In questo capitolo, avete imparato a conoscere le funzioni interne di Python, come `float` e `int`, che possono convertire numeri con il punto decimale in interi e viceversa. Avete visto anche come la funzione `len` possa rendere più facili i cicli e come si possa usare Python per aprire file per leggerne i contenuti o per scrivere qualcosa al loro interno.

ROMPICAPO DI PROGRAMMAZIONE

Provate a realizzare gli esempi seguenti, per sperimentare le funzioni interne di Python. Trovate le risposte all’indirizzo *http://python-for-kids.com/*.

1. IL CODICE MISTERIOSO

Quale sarà il risultato dell’esecuzione di questo codice? Provate a ricostruirlo a mente, poi eseguite il codice per vedere se avete trovato la risposta giusta.

```
>>> a = abs(10) + abs(-10)
>>> print(a)
>>> b = abs(-10) + -10
>>> print(b)
```

2. IL MESSAGGIO NASCOSTO

Provate a usare `dir` e `help` per scoprire come suddividere una stringa in parole, poi create un programmino per stampare una parola sì e una no della stringa seguente, partendo con la prima parola (questo):

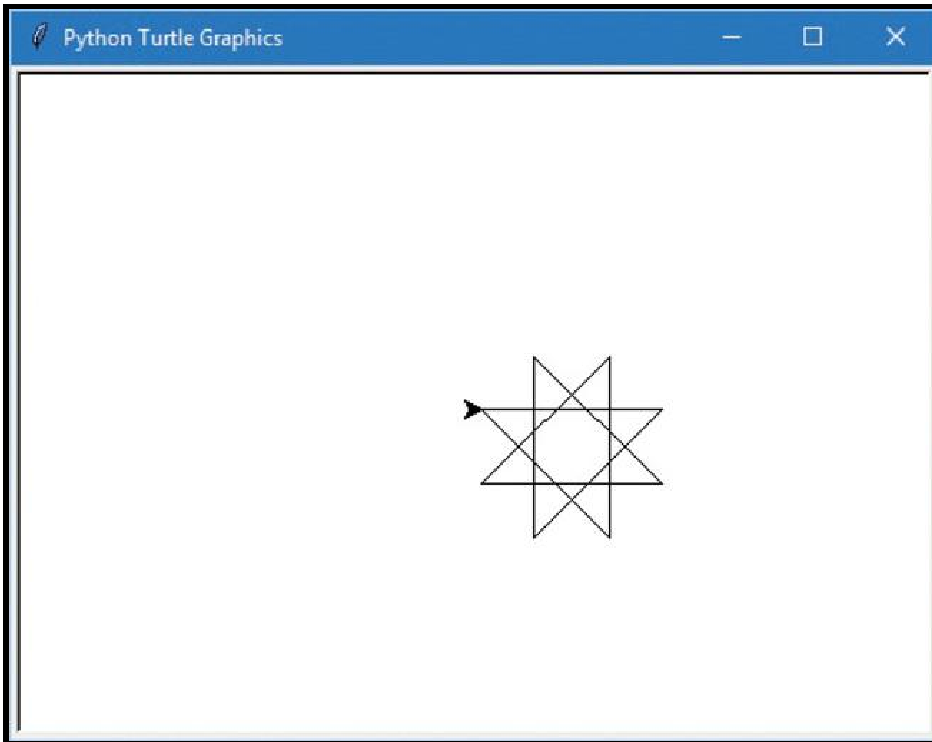
```
"questo se non tu è leggendo un questo modo cominciando molto
azzurro furbo sei per sette nascondere venti un rosso messaggio
tu"
```

3. COPIARE UN FILE

Create un programma in Python per copiare un file. (Suggerimento: dovete aprire il file che volete copiare, leggerlo e poi creare un nuovo file, la copia.) Verificate che il vostro programma funzioni visualizzando sullo schermo i contenuti del nuovo file.

```
>>> t.reset()
>>> for x in range(1, 9):
        t.forward(100)
        t.left(225)
```

Questo codice genera una stella a otto punte:



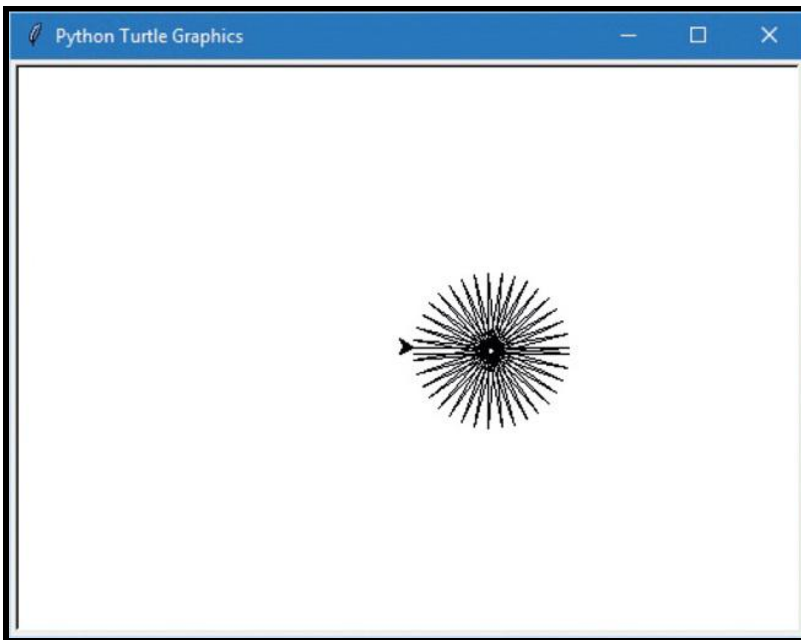
Il codice in sé è molto simile a quello per disegnare il quadrato, con qualche eccezione:

- Anziché eseguire il ciclo quattro volte, con `range(1, 5)`, lo esegue otto volte con `range(1, 9)`.
- Anziché avanzare di 50 pixel, la tartaruga avanza di 100.
- Anziché ruotare di 90 gradi, la tartaruga ruota di 225 gradi a sinistra.

Proviamo a sviluppare ancora un po’ la nostra stella. Utilizzando un angolo di 175 gradi e percorrendo il ciclo 37 volte, possiamo creare una stella con un numero ancora maggiore di punte:

```
>>> t.reset()
>>> for x in range(1, 38):
        t.forward(100)
        t.left(175)
```

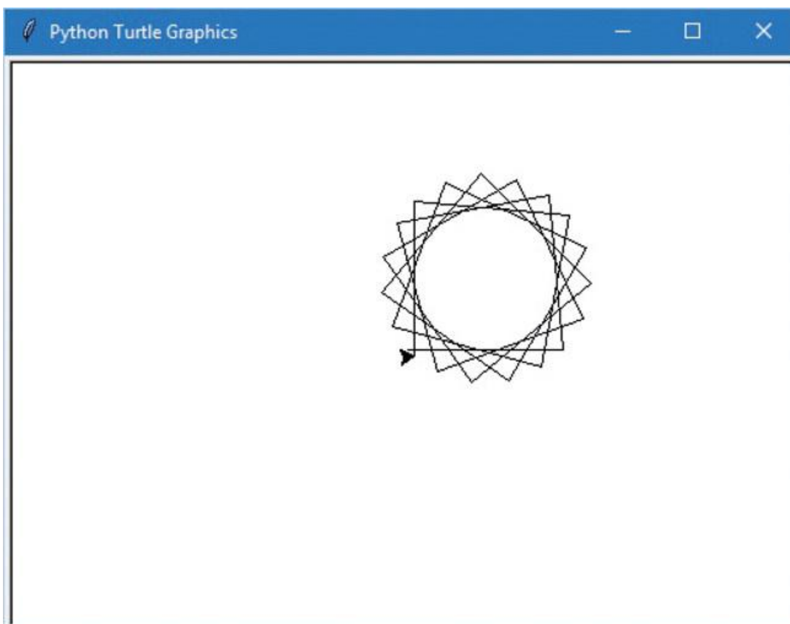
Ecco il risultato:



Restando in tema di stelle, ecco come produrre una stella “a spirale”:

```
>>> t.reset()
>>> for x in range(1, 20):
        t.forward(100)
        t.left(95)
```

Modificando l’angolo di rotazione e riducendo il numero dei cicli, la tartaruga disegna un tipo di stella molto diverso:



Utilizzando un codice simile, possiamo creare varie forme, da un semplice quadrato a una stella a spirale. Come potete vedere, grazie ai cicli `for`, abbiamo reso molto più semplice il disegno di queste forme. Senza i cicli, sarebbe stato necessario scrivere una gran quantità di codice noiosamente ripetitivo.

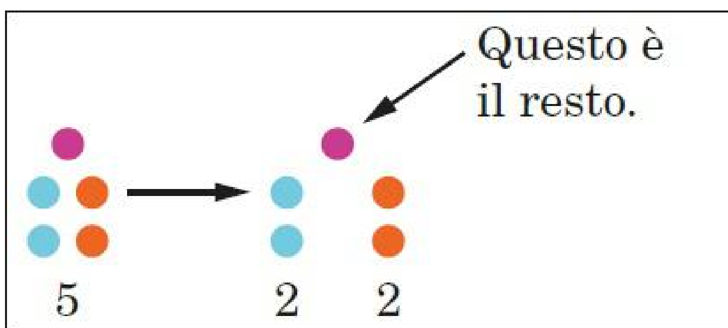
Ora, proviamo a usare un enunciato `if` per controllare la rotazione della

tartaruga e disegnare un'altra variante di stella. In questo esempio, vogliamo che la tartaruga ruoti di un certo angolo la prima volta, poi di un angolo diverso la volta successiva.

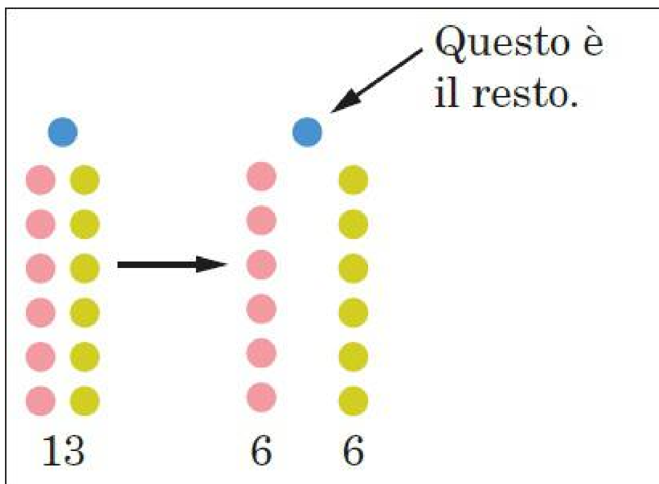
```
>>> t.reset()
>>> for x in range(1, 19):
        t.forward(100)
        if x % 2 == 0:
                t.left(175)
        else:
                t.left(225)
```

Qui abbiamo creato un ciclo, che viene eseguito 18 volte (`range(1, 19)`) e dice alla tartaruga di spostarsi in avanti di 100 pixel (`t.forward(100)`). A questo punto arriva l'enunciato `if x % 2 == 0`, che controlla se la variabile `x` contiene un numero pari, utilizzando un operatore *modulo*, indicato dal simbolo `%` nell'espressione `x % 2 == 0`, che sostanzialmente significa "x mod 2" è uguale a 0.

L'operatore modulo dà il resto della divisione di due numeri, in questo caso il resto della divisione del valore contenuto nella variabile `x` per 2. Per esempio, se dividessimo 5 palline in due gruppi, otterremmo due gruppi di 2 palline per un totale di 4 palline) e un resto costituito da 1 pallina:



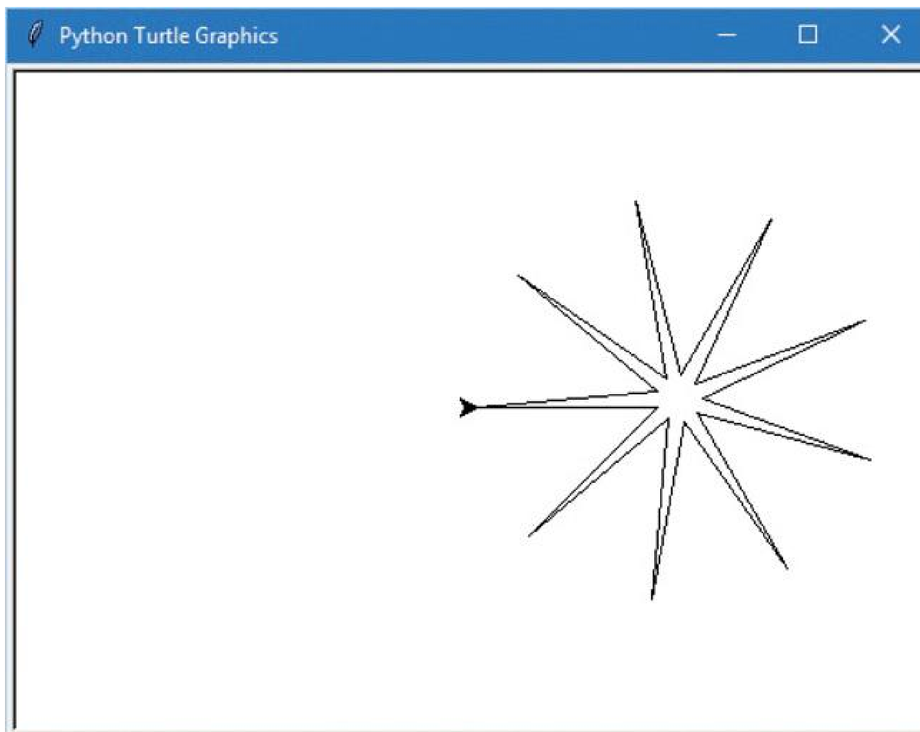
Se dividessimo 13 palline in due parti, otterremmo due gruppi di 6 palline con il resto di 1 pallina:



Quando controlliamo se il resto è uguale a zero, dopo aver diviso per 2, sostanzialmente ci chiediamo se sia possibile dividere il valore che ci interessa in due parti senza resto. Questo è un metodo elegante per sapere se il numero contenuto in una variabile è pari: la divisione per 2 di un numero pari dà sempre resto 0, mentre la divisione per 2 di un numero dispari dà sempre resto 1 (diverso da zero).

Nella quinta riga del nostro codice, diciamo alla tartaruga di ruotare a sinistra di 175 gradi (`t.left(175)`) se il numero in `x` è pari (`if x % 2 == 0:`); altrimenti (`else`), nell'ultima riga, le diciamo di ruotare di 225 gradi (`t.left(225)`).

Ecco il risultato:



DISEGNARE UN'AUTO

La tartaruga può fare molto più che disegnare stelle e semplici forme geometriche. Nel prossimo esempio, disegneremo un'auto, sia pure di forma un po' rozza.

Prima, disegniamo il corpo della vettura. In IDLE, selezionate **File** > **New File**, poi inserite nella nuova finestra questo codice:

```
import turtle
t = turtle.Pen()
t.reset()
t.color(1,0,0)
t.begin_fill()
t.forward(100)
t.left(90)
t.forward(20)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(60)
t.left(90)
t.forward(20)
t.right(90)
t.forward(20)
t.left(90)
t.forward(20)
t.end_fill()
```

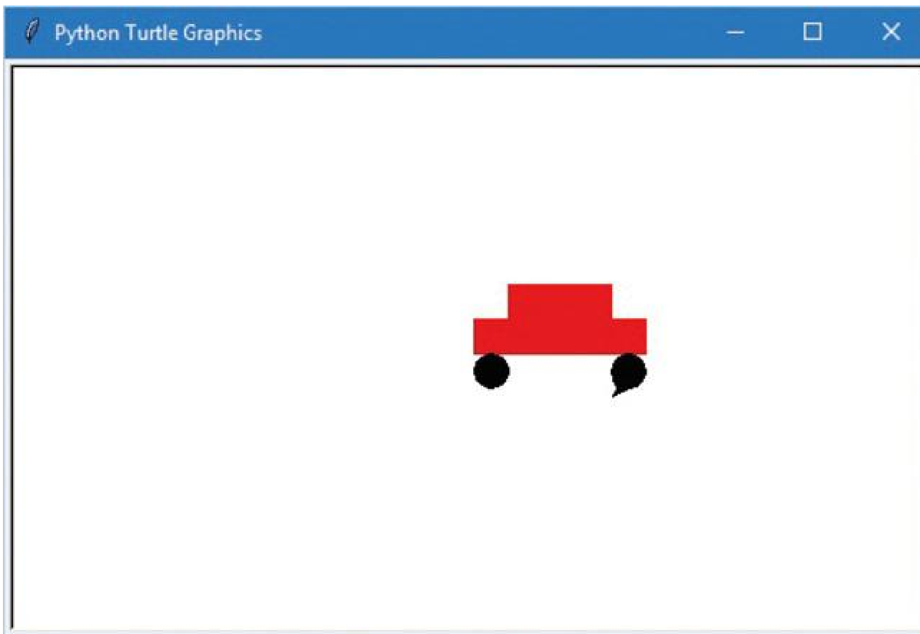
Poi, disegniamo la prima ruota.

```
t.color(0,0,0)
t.up()
t.forward(10)
t.down()
t.begin_fill()
t.circle(10)
t.end_fill()
```

Infine, la seconda ruota:

```
t.setheading(0)
t.up()
t.forward(90)
t.right(90)
t.forward(10)
t.setheading(0)
t.begin_fill()
t.down()
t.circle(10)
t.end_fill()
```

Selezionate **File > Save As** e date al file un nome come *auto.py*. Selezionate poi **Run > Run Module** per provare il codice. Ed ecco la vostra auto:



Avrete probabilmente notato che nel nostro codice è comparsa qualche nuova funzione:

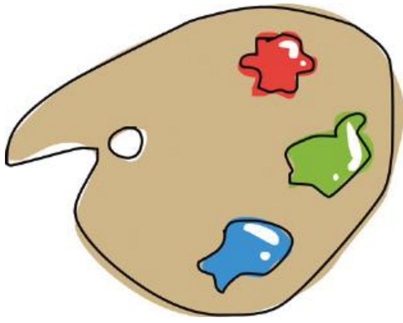
- `color` è usata per cambiare il colore della penna.
- `begin_fill` e `end_fill` si usano per riempire di colore un'area della tela.
- `circle` disegna un cerchio di una particolare dimensione.
- `setheading` fa ruotare la tartaruga in modo che punti in una certa direzione.

Diamo un'occhiata a come si possono usare queste funzioni per aggiungere un po' di colore ai disegni.

COLORARE

La funzione `color` prende tre parametri. Il primo specifica la quantità di rosso, il secondo la quantità di verde, il terzo la quantità di blu. Per esempio, per avere un'auto di un rosso acceso, abbiamo usato `color(1,0,0)`, che dice alla tartaruga di usare una penna rossa al 100 per cento.

La combinazione di rosso, verde e blu per ottenere tutti i colori prende il nome di *RGB* (dall'inglese *red*, *green*, *blue*). È il modo in cui sono rappresentati i colori sui monitor e le diverse combinazioni di questi colori primari producono altri colori, come quando si mescolano vernici blu e rosse per ottenere il viola o gialle e rosse per ottenere l'arancione. I colori rosso, verde e blu sono chiamati *primari* perché non si possono ottenere mescolando altri colori.



Anche se non usiamo vernici quando creiamo colori su un monitor (usiamo la luce), può essere utile pensare queste “ricette” RGB come formate da tre latte di vernice, una rossa, una verde e una blu. Ogni lattina è piena e alla lattina piena attribuiamo un valore 1 (o 100 per cento). Poi mescoliamo tutta la vernice rossa e tutta quella verde in un contenitore, ottenendo il giallo (1 e 1, 100 per cento di ciascun colore).

Torniamo al mondo del codice. Per disegnare un cerchio giallo con la tartaruga, dovremmo usare il 100 per cento di vernice rossa e di vernice verde, ma niente blu, così:

```
>>> t.color(1,1,0)
>>> t.begin_fill()
>>> t.circle(50)
>>> t.end_fill()
```

La terna `1,1,0` nella prima riga rappresenta il 100 per cento di rosso, il 100 per cento di verde e lo 0 per cento di blu. Nella riga successiva, diciamo alla tartaruga di riempire con questo colore RGB le forme che disegna `(t.begin_fill)`, poi di disegnare un cerchio con `t.circle`. Nell’ultima riga, `end_fill` dice alla tartaruga di riempire il cerchio con il colore RGB.

UNA FUNZIONE PER UN CERCHIO PIENO

Per facilitarci la sperimentazione con vari colori, creiamo una funzione dal codice utilizzato per disegnare un cerchio pieno.

```
>>> def miocerchio(rosso, verde, blu):
        t.color(rosso, verde, blu)
        t.begin_fill()
        t.circle(50)
        t.end_fill()
```

Possiamo disegnare un cerchio di colore verde brillante utilizzando solo la vernice verde, con questo codice:

```
>>> miocerchio(0, 1, 0)
```

Oppure possiamo disegnare un cerchio di verde più scuro usando solo metà della vernice verde (0.5):

```
>>> miocerchio(0, 0.5, 0)
```

Per giocare con i colori RGB sullo schermo, provate a disegnare un cerchio prima con il 100 per cento e poi con il 50 per cento di rosso (1 e 0.5), poi con il solo blu, al 100 e poi al 50 per cento:

```
>>> miocerchio(1, 0, 0)
>>> miocerchio(0.5, 0, 0)
>>> miocerchio(0, 0, 1)
>>> miocerchio(0, 0, 0.5)
```

NOTA

Se la tela comincia a essere un po' ingombra, usate `t.reset()` *per cancellare tutti i vecchi disegni. Ricordate inoltre che potete spostare la tartaruga senza disegnare nulla utilizzato* `t.up()` *per alzare la penna, per poi riabbassarla con* `t.down()`.

Varie combinazioni di rosso, verde e blu produrranno tanti colori diversi, come il color oro:

```
>>> miocerchio(0.9, 0.75, 0)
```

Questo è un rosa:

```
>>> miocerchio(1, 0.7, 0.75)
```

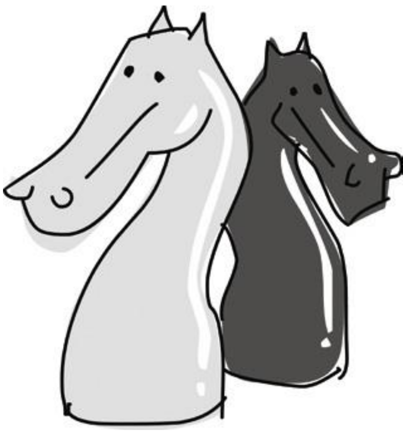
Ed ecco due diverse sfumature di arancione:

```
>>> miocerchio(1, 0.5, 0)
>>> miocerchio(0.9, 0.5, 0.15)
```

Provate a realizzare i colori che preferite!

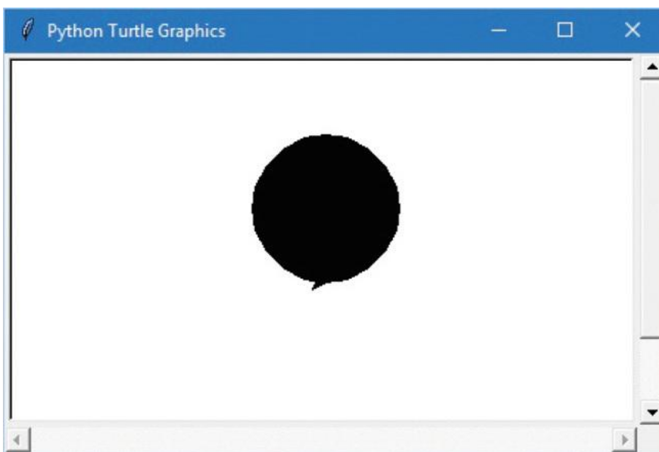
CREARE UN BIANCO E NERO PURO

Che cosa succede quando si spengono tutte le luci la sera? Tutto diventa nero. Lo stesso succede con i colori di un computer. Nessuna luce significa nessun colore, così un cerchio con 0 per tutti i colori primari risulterà riempito di nero:



```
>>> miocerchio(0, 0, 0)
```

Ecco il risultato:



Succede il contrario se si usa il 100 per cento di tutti e tre i colori: in quel caso, si ottiene il bianco. Scrivete questa istruzione, per cancellare il cerchio nero:

```
>>> miocerchio(1, 1, 1)
```

UNA FUNZIONE PER DISEGNARE QUADRATI

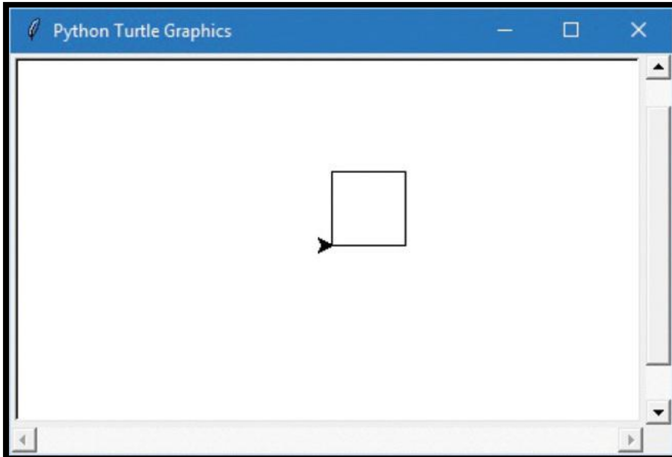
Abbiamo visto che si riempiono di colore le forme dicendo alla tartaruga di iniziare il riempimento con `begin_fill`, e poi le forme vengono riempite quando si usa la funzione `end_ fill`. Ora facciamo qualche altro esperimento con le forme e i riempimenti. Usiamo la funzione che disegna quadrati (vista all'inizio del capitolo) e le passiamo il lato del quadrato come parametro.

```
>>> def mioquadrato(lato):
        for x in range(1, 5):
            t.forward(lato)
            t.left(90)
```

Provate la funzione chiamandola con un lato di 50 pixel:

```
>>> t.reset()
>>> mioquadrato(50)
```

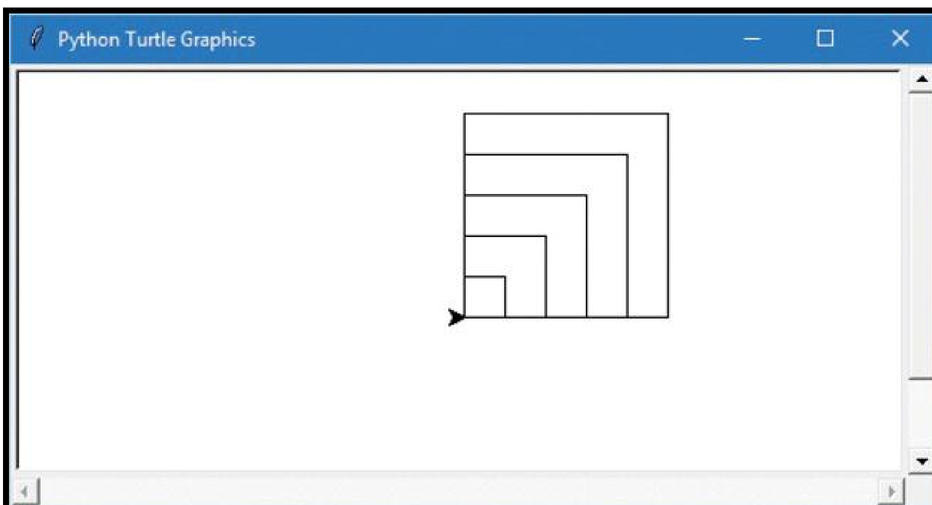
Questo produrrà un piccolo quadrato:



Ora proviamo con lati di misura diversa. Creiamo cinque quadrati consecutivi di lato 25, 50, 75, 100 e 125.

```
>>> t.reset()
>>> mioquadrato(25)
>>> mioquadrato(50)
>>> mioquadrato(75)
>>> mioquadrato(100)
>>> mioquadrato(125)
```

Questi quadrati saranno fatti in questo modo:



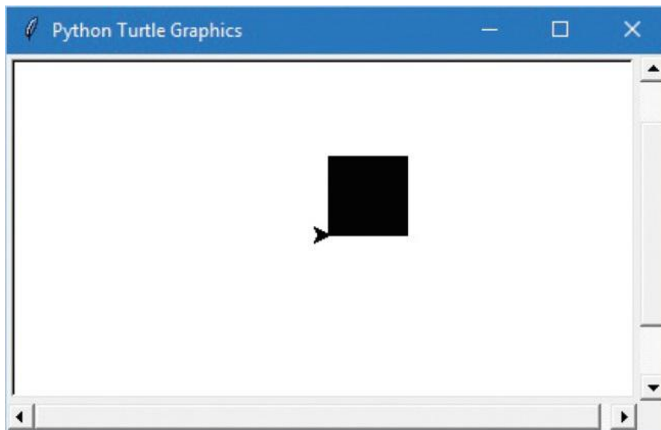
DISEGNARE QUADRATI PIENI

Per disegnare un quadrato pieno, bisogna prima reinizializzare la tela, iniziare il riempimento e poi chiamare di nuovo la funzione `quadrato`, con questo codice:

```
>>> t.reset()
>>> t.begin_fill()
>>> mioquadrato(50)
```

Vedrete un quadrato vuoto, fino a che non finite il riempimento:

```
>>> t.end_fill()
```



Cambiamo questa funzione, in modo poter disegnare quadrati pieni o vuoti. Abbiamo bisogno di un altro parametro, e il codice diventa un po’ più complesso:

```
>>> def mioquadrato(lato, pieno):
        if pieno == True:
                t.begin_fill()
        for x in range(1, 5):
                t.forward(lato)
                t.left(90)
        if pieno == True:
                t.end_fill()
```

Nella prima riga, cambiamo la definizione della funzione, in modo che prenda due parametri `lato` e `pieno`. Poi vediamo se il valore di `pieno` è `True`, con `if pieno == True`. Se lo è, chiamiamo `begin_fill`, per dire alla tartaruga di riempire la forma che disegneremo. Poi effettuiamo per quattro volte (`for x in range(1, 5)`) il ciclo per disegnare i quattro lati del quadrato (procedendo in avanti e a sinistra), prima di verificare di nuovo se `pieno` è `True` con `if pieno = True`. Se lo è, completiamo il riempimento con `t.end_fill()` e la tartaruga riempie di colore il quadrato.

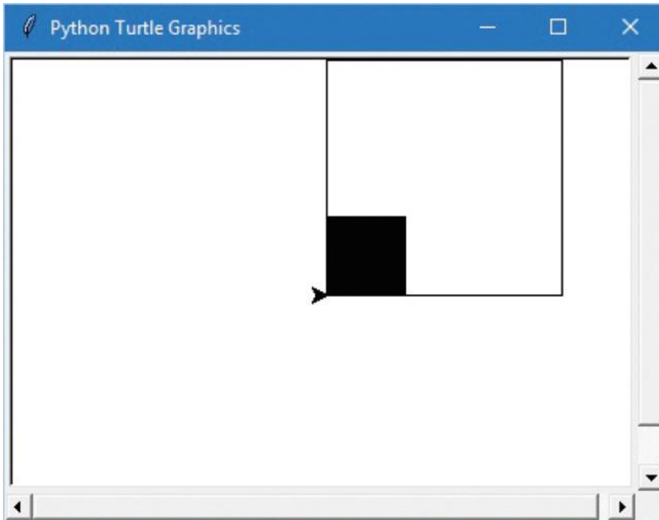
Ora possiamo disegnare un quadrato pieno con questo codice:

```
>>> mioquadrato(50, True)
```

Oppure possiamo creare un quadrato vuoto con questo:

```
>>> mioquadrato(150, False)
```

Dopo queste due chiamate alla funzione `mioquadrato`, si ottiene questa immagine, che sembra un po' un occhio quadrato.



Ovviamente non ci si ferma qui: è possibile disegnare e riempire di colore ogni tipo di forma.

DISEGNARE STELLE PIENE

Come ultimo esempio, aggiungeremo un po' di colore alla stella che abbiamo disegnato prima. Il codice originale era questo:

```
>>> for x in range(1, 19):
        t.forward(100)
        if x % 2 == 0:
                t.left(175)
        else:
                t.left(225)
```

Su questa base, costruiamo ora una funzione `miastella`. Useremo gli enunciati `if` della funzione `mioquadrato` e aggiungeremo il parametro `lato`.

```
>>> def miastella(lato, pieno):
        if pieno == True:
                t.begin_fill()
        for x in range(1, 19):
                t.forward(lato)
                if x % 2 == 0:
                        t.left(175)
                else:
                        t.left(225)
        if pieno == True:
                t.end_fill()
```

Nelle prime due righe di questa funzione, verifichiamo se `pieno` è `True`, e se lo è iniziamo il riempimento. Verifichiamo ancora nelle ultime due righe e, se `pieno` è `True`, concludiamo il riempimento. Come nella funzione `mioquadrato`, passiamo il lato della stella nel parametro `lato` e usiamo quel valore quando chiamiamo `t.forward`.

Ora reinizializziamo il canvas, impostiamo il colore a dorato (90 per cento di rosso, 75 per cento di verde e 0 per cento di blu), poi chiamiamo la funzione.

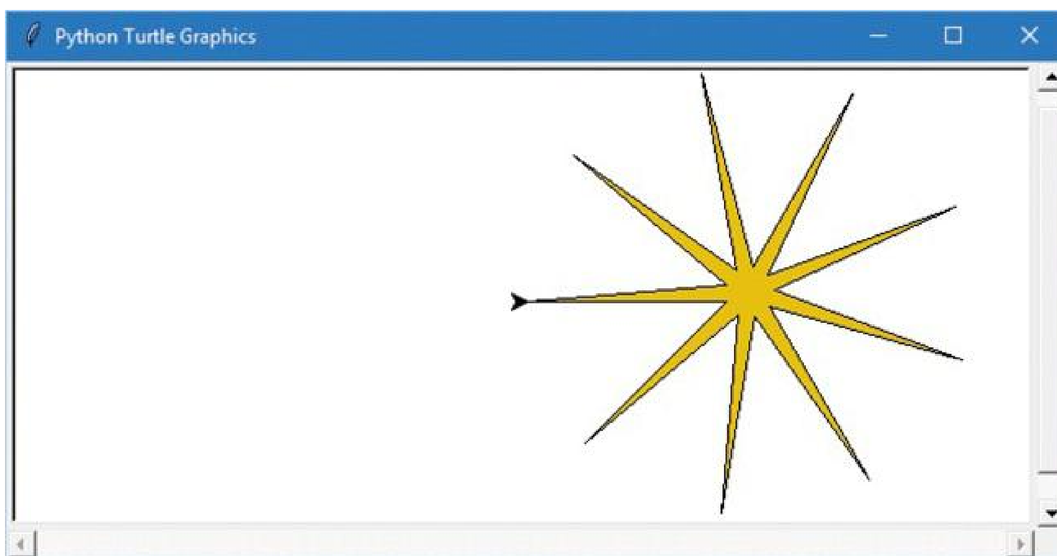
```
>>> t.reset()
>>> t.color(0.9, 0.75, 0)
>>> miastella(120, True)
```

La tartaruga disegnerà la stella colorata:

Per aggiungere un contorno alla stella, possiamo cambiare il colore in nero e ridisegnare la stella senza riempimento:

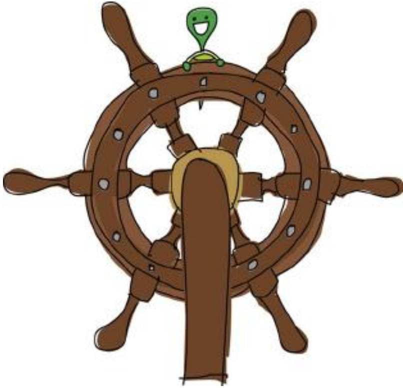
```
>>> t.color(0, 0, 0)
>>> miastella(120, False)
```

E ora abbiamo una stella dorata con un contorno in nero:



CHE COSA AVETE IMPARATO

In questo capitolo, avete imparato come usare il modulo `turtle` per disegnare alcune forme geometriche di base, mediante i cicli `for` e gli enunciati `if` per controllare quello che fa la tartaruga. Abbiamo modificato il colore della penna e riempito le forme disegnate. Abbiamo anche riusato il codice di disegno per creare alcune funzioni che semplificano il disegno di forme con colori diversi con un'unica chiamata a una funzione.

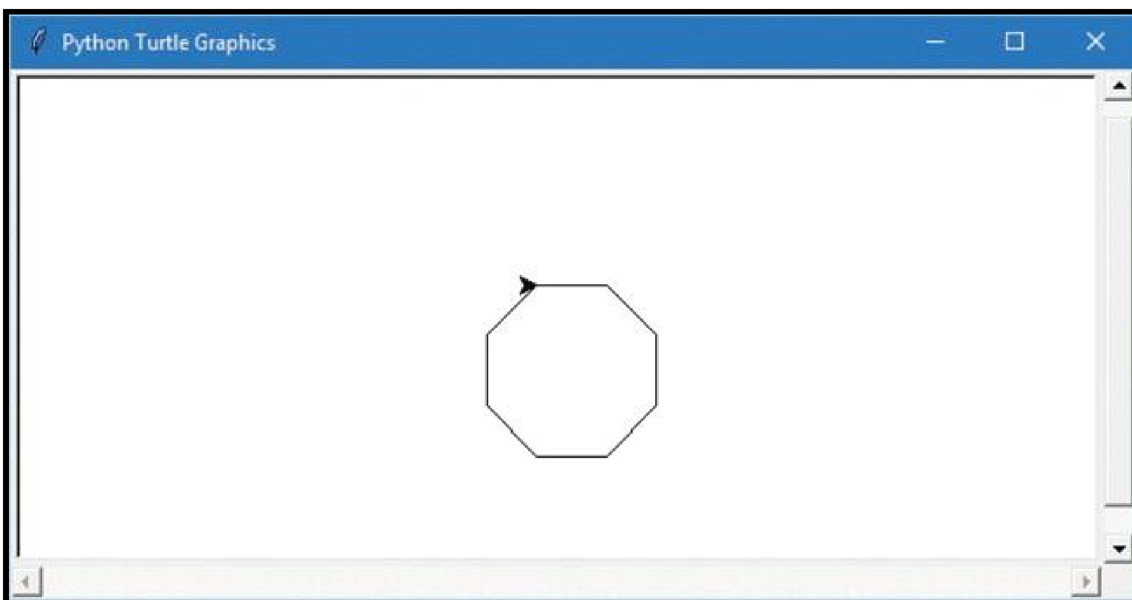


ROMPICAPO DI PROGRAMMAZIONE

Negli esperimenti seguenti, dovrete disegnare qualche forma con la tartaruga. Come sempre, le soluzioni si possono trovare all'indirizzo *http://python-for-kids.com*.

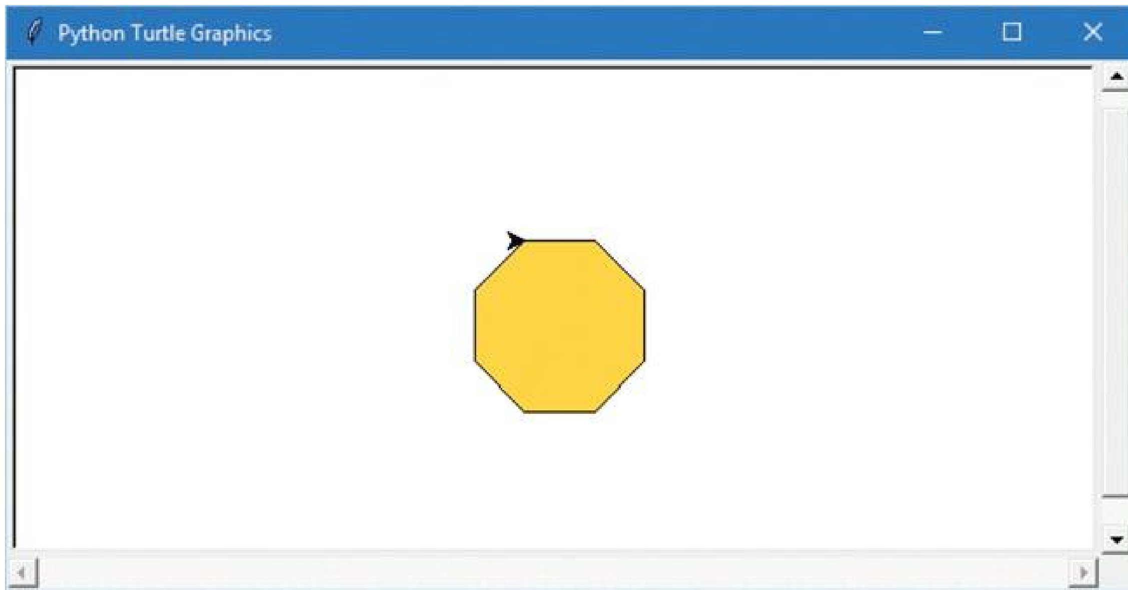
1. DISEGNARE UN OTTAGONO

In questo capitolo abbiamo disegnato stelle, quadrati e rettangoli. Che cosa ne dite di creare una funzione che disegni una forma a otto lati, come un ottagono? (Suggerimento: provate a far ruotare la tartaruga di 45 gradi.)



2. DISEGNARE UN OTTAGONO PIENO

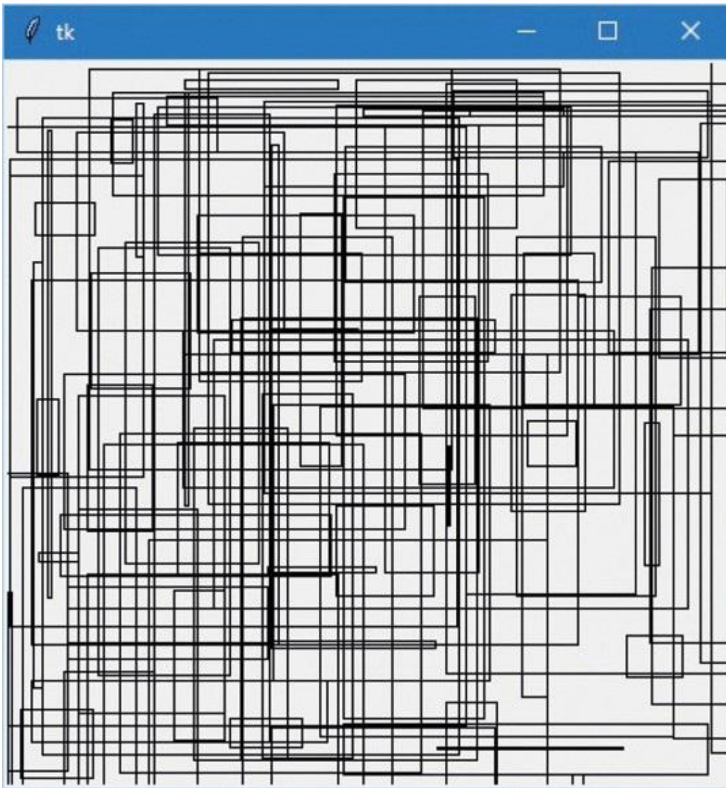
Ora che avete una funzione che disegna un ottagono, modificatela in modo che disegni un ottagono pieno. Provate a disegnare un ottagono con un contorno, come abbiamo fatto con la stella.



3. UN’ALTRA FUNZIONE PER DISEGNARE STELLE

Create una funzione per disegnare una stella che prenda due parametri: il lato e il numero delle punte. L’inizio della funzione sarà analogo a questo:

```
def disegna_stella(lato, punte):
```

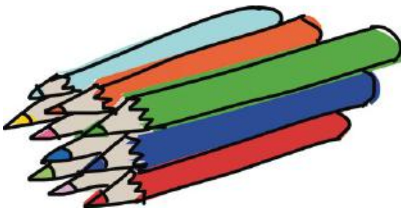


IMPOSTARE IL COLORE

Diamo un po' di colore alla nostra grafica. Modifichiamo la funzione `random_rectangle` in modo da passarle un colore per il rettangolo come altro parametro (`fill_color`). Inserite questo codice in un nuovo file e salvatelo con il nome *colorrect.py*:

```
from tkinter import *
import random
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
def random_rectangle(width, height, fill_color):
        x1 = random.randrange(width)
        y1 = random.randrange(height)
        x2 = random.randrange(x1 + random.randrange(width-x1))
        y2 = random.randrange(y1 + random.randrange(height-y1))
        canvas.create_rectangle(x1, y1, x2, y2, fill=fill_color)
```

La funzione `create_rectangle` ora prende un parametro `fill_color`, che specifica il colore da usare nel disegnare il rettangolo.



Possiamo passare alla funzione i colori per nome, come nell'esempio che segue (utilizzando una tela di 400 pixel per 400 pixel) in modo da creare una serie di rettangoli di colore diverso. Se volete provare questo esempio, potete copiare e

incollare per evitare di riscrivere molte volte gli stessi caratteri. Selezionate il testo da copiare, premete CTRL+C per copiarlo, fate clic su una riga vuota e premete CTRL+V per incollare. Scrivete la prima riga, poi la copiate, la incollate e modificate solo la parte che varia. Aggiungete questo codice a *colorrect.py*, subito sotto la funzione):

```
random_rectangle(400, 400, 'green')
random_rectangle(400, 400, 'red')
random_rectangle(400, 400, 'blue')
random_rectangle(400, 400, 'orange')
random_rectangle(400, 400, 'yellow')
random_rectangle(400, 400, 'pink')
random_rectangle(400, 400, 'purple')
random_rectangle(400, 400, 'violet')
random_rectangle(400, 400, 'magenta')
random_rectangle(400, 400, 'cyan')
```

Molti di questi colori indicati per nome visualizzeranno i colori che vi aspettate, ma altri potrebbero produrre un messaggio di errore (a seconda che usiate Windows, Mac OS X o Linux).

Ma se volessimo un colore speciale che non ha un nome? Ricordate dal Capitolo 11 che abbiamo impostato il colore della penna della tartaruga usando percentuali dei colori rosso, verde e blu. Impostare la quantità di ciascun colore primario (rosso, verde e blu) per ottenere una combinazione di colore in `tkinter` è leggermente più complicato.

Lavorando con il modulo `turtle` abbiamo creato il colore dorato usando il 90 per cento di rosso, il 75 per cento di verde e niente blu. In `tkinter`, si può creare lo stesso colore così:

```
random_rectangle(400, 400, '#ffd800')
```

Il segno `#` (cancelletto o diesis) davanti al valore `ffd800` dice a Python che quello che segue è un numero esadecimale. L'esadecimale è un sistema di rappresentazione dei numeri utilizzato comunemente nella programmazione informatica. Usa una base 16 (come simboli si usano le cifre da 0 a 9 e quindi le lettere da A a F) anziché la base 10 (simboli le cifre da 0 a 9) del sistema decimale. Se non avete ancora studiato le basi in matematica, basta sappiate che si può convertire un normale numero decimale in esadecimale utilizzando un *segnaposto di formato* in una stringa: `%x` (tornate a vedere "Incorporare valori nelle stringhe" a pagina 30). Per esempio, per convertire il numero decimale 15 in esadecimale, potete fare così:

```
>>> print('%x' % 15)
f
```

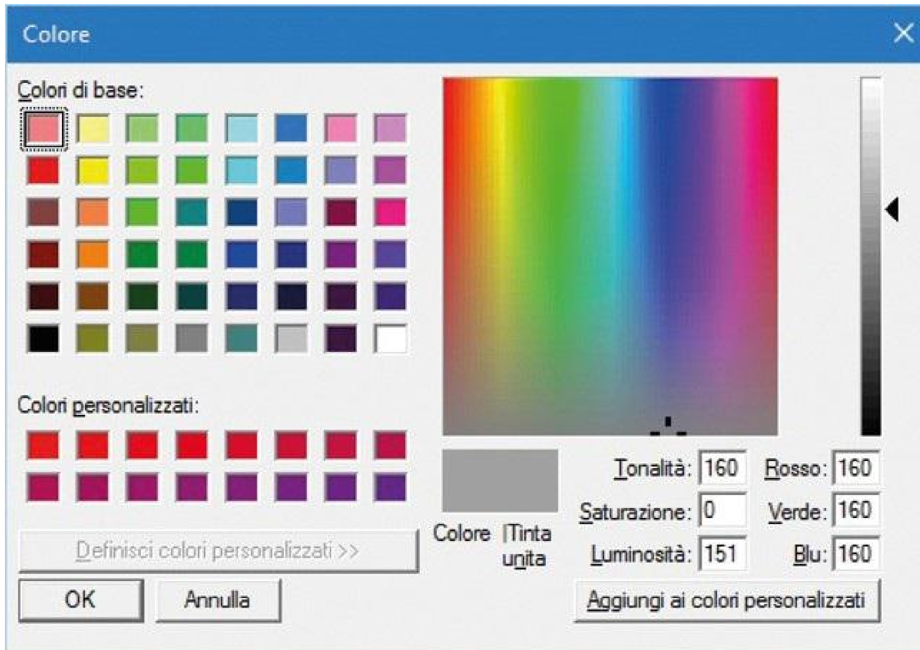
Per essere sicuri che il nostro numero abbia almeno due cifre, possiamo modificare leggermente il segnaposto di formato in questo modo:

```
>>> print('%02x' % 15)
0f
```

Il modulo `tkinter` mette a disposizione anche un modo facile per ottenere un valore esadecimale per un colore. Provate ad aggiungere questo codice a *colorrect.py* (potete eliminare le altre chiamate alla funzione `random_rectangle`).

```
from tkinter import *
colorchooser.askcolor()
```

Vi verrà presentata una finestra per la scelta del colore.



Se, anziché da file, date il comando dalla shell, selezionate un colore e fate clic su OK, verrà visualizzata una tupla, che contiene una tupla con tre numeri e una stringa:

```
>>> colorchooser.askcolor()
((135.52734375, 186.7265625, 165.64453125), '#87baa5')
```

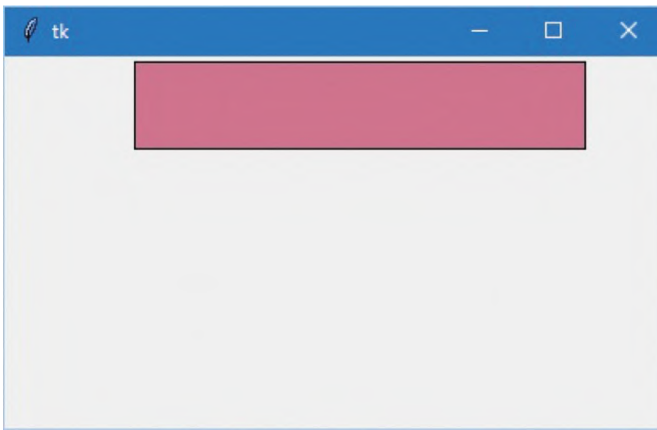
I tre numeri rappresentano le quantità di rosso, verde e blu. In `tkinter`, la quantità di ciascun colore da usare in un colore composto è rappresentata da un numero compreso fra 0 e 255 (che è diverso dall'usare una percentuale per ciascun colore primario con il modulo `turtle`). La stringa nella tupla contiene la versione esadecimale di questi tre numeri.

Potete copiare e incollare il valore stringa da utilizzare oppure memorizzare la tupla come una variabile, poi usare l'indice posizionale del valore esadecimale.

Proviamo a usare la funzione `random_rectangle` per vedere come funziona:

```
>>> c = colorchooser.askcolor()
>>> random_rectangle(400, 400, c[1])
```

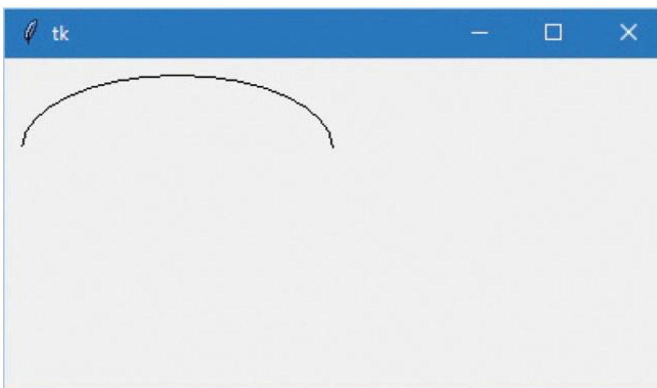
Ed ecco il risultato:



DISEGNARE ARCHI

Un arco è una porzione di circonferenza o di qualche altra curva, ma per poterne disegnare uno con `tkinter`, bisogna disegnarlo all'interno di un rettangolo con la funzione `create_arc`, per esempio così:

```
canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

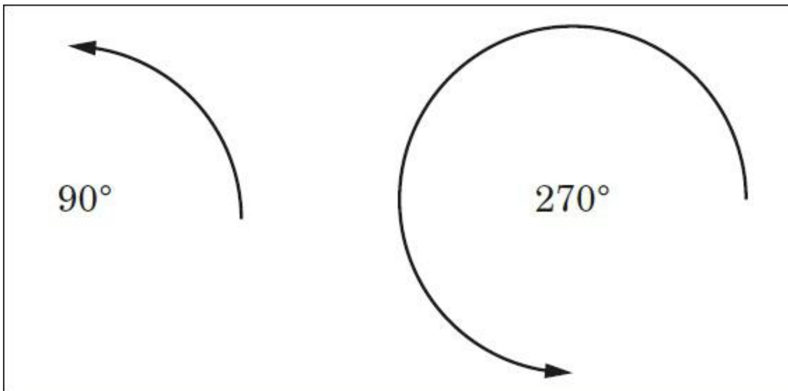


Se avete chiuso tutte le finestre `tkinter`, o avete riavviato IDLE, ricordate di reimportare `tkinter` e di ricreare la tela:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 100, extent=180, style=ARC)
```

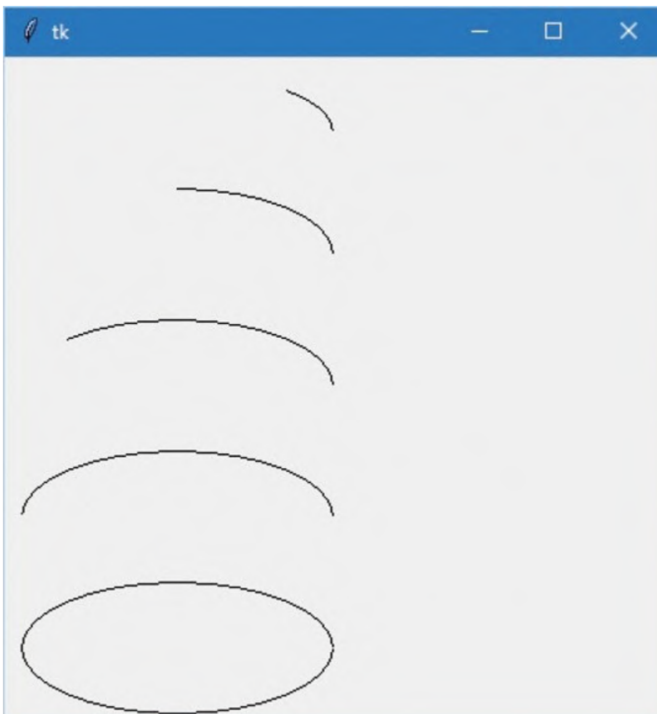
Questo codice posiziona il vertice superiore sinistro del rettangolo che conterrà l'arco alle coordinate (10, 10), cioè a 10 pixel dal lato sinistro e a 10 pixel dal lato superiore della finestra, e il vertice inferiore destro alle coordinate (200, 100). Il

parametro successivo, `extent`, si usa per specificare i gradi dell'arco. Ricordate dal Capitolo 4 che i gradi sono un modo per misurare la distanza percorsa su una circonferenza. Ecco due esempi di archi, che corrispondono a uno spostamento rispettivamente di 90 e di 270 gradi su una circonferenza:



Il codice seguente disegna diversi archi, in modo che possiate vedere che cosa succede utilizzando un numero di gradi diverso con la funzione `create_arc`.

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_arc(10, 10, 200, 80, extent=45, style=ARC)
>>> canvas.create_arc(10, 80, 200, 160, extent=90, style=ARC)
>>> canvas.create_arc(10, 160, 200, 240, extent=135, style=ARC)
>>> canvas.create_arc(10, 240, 200, 320, extent=180, style=ARC)
>>> canvas.create_arc(10, 320, 200, 400, extent=359, style=ARC)
```



NOTA

Nell'ultimo caso abbiamo usato 359 gradi anziché 360, perché per `tkinter` *360 è*

uguale a 0 gradi, e se avessimo usato 360 non avrebbe disegnato nulla.

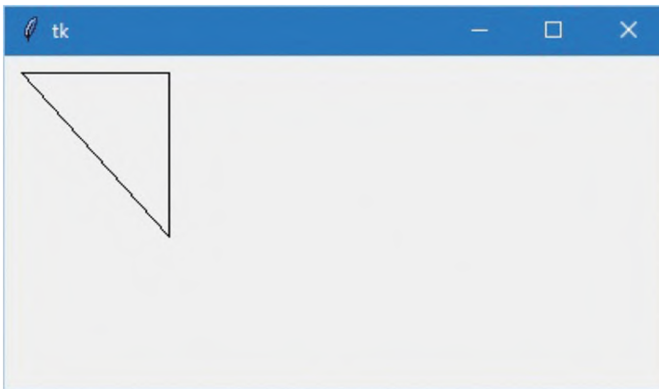
DISEGNARE POLIGONI

Un poligono è una figura chiusa con tre o più lati. Esistono poligoni regolari come triangoli, quadrati, rettangoli, pentagoni, esagoni e così via, e anche poligoni irregolari con lati diversi, molti più lati, e forme strane.

Per disegnare poligoni con `tkinter`, bisogna dare le coordinate di ciascun punto del poligono. Ecco come si può disegnare un triangolo:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 100, 10, 100, 110, fill="",
outline="black")
```

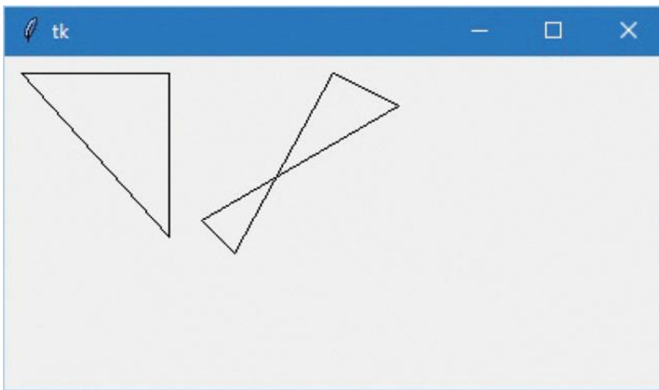
L’esempio disegna un triangolo partendo dalle coordinate x e y (10, 10), per poi passare alle coordinate (100, 10) e finire in (100, 110). Qui a destra si vede il risultato.



Possiamo aggiungere un altro poligono irregolare (una forma con angoli o lati diversi) con questo codice:

```
>>> canvas.create_polygon(200, 10, 240, 30, 120, 100, 140, 120,
fill="", outline="black")
```

Qui si parte con le coordinate (200, 10), poi si passa a (240, 30), poi a (120, 100) e infine a (140, 120). `Tkinter` disegna automaticamente l’ultimo lato, che congiunge il punto indicato dall’ultima coppia di coordinate con quello indicato dalla prima coppia. Ed ecco il risultato dell’esecuzione di questo codice:



VISUALIZZARE TESTO

Oltre a disegnare forme, si può anche scrivere sul canvas con la funzione `create_text`. Questa funzione prende solo due coordinate (le posizioni x e y del testo), insieme con il parametro per nome del testo da visualizzare. Nel codice che segue, creiamo la tela, poi visualizziamo una frase alle coordinate (150, 100). Salvatelo in un file con il nome *testo.py*.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=600, height=400)
canvas.pack()
canvas.create_text(150, 100, text="Signor maestro, che le salta
in mente?")
```

La funzione `create_text` prende anche alcuni altri parametri utili, come un colore di riempimento del testo. Nel codice che segue, chiamiamo la funzione `create_text` con le coordinate (130, 120), il testo che vogliamo visualizzare e un colore di riempimento rosso.

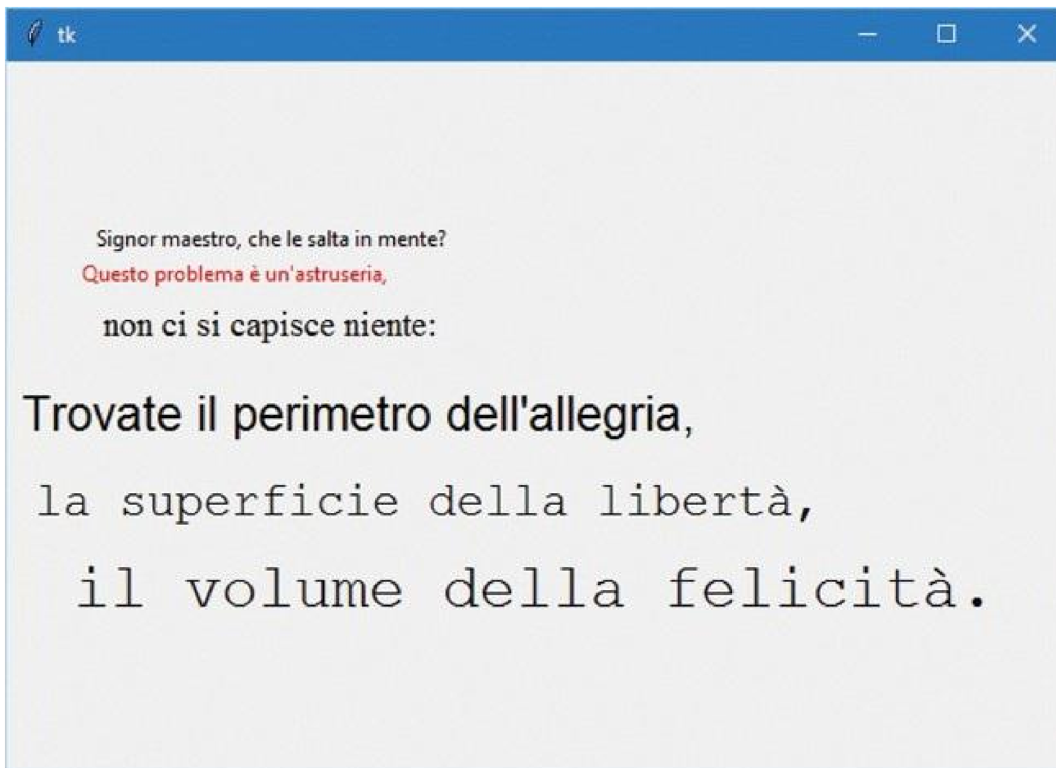
```
canvas.create_text(130, 120, text="Questo problema è
un'astruseria,", fill='red')
```

Si può anche specificare la font (il tipo di carattere usato per visualizzare il testo) come una tupla con il nome della font e il corpo (dimensione) del testo. Per esempio, la tupla per il Times corpo 20 è (`'Times', 20`). Nel codice seguente, visualizziamo il testo volta a volta in Times corpo 15, Helvetica corpo 20, Courier corpo 20 e poi 26.

```
canvas.create_text(150, 150, text="non ci si capisce niente:",
font=('Times', 15))
canvas.create_text(200, 200, text="Trovate il perimetro
dell'allegria,", font=('Helvetica', 20))
canvas.create_text(240, 250, text="la superficie della libertà,",
font=('Courier', 20))
canvas.create_text(300, 300, text="il volume della felicità.",
font=('Courier', 26))
```

Ed ecco il risultato di queste funzioni con le tre font specificate e le diverse

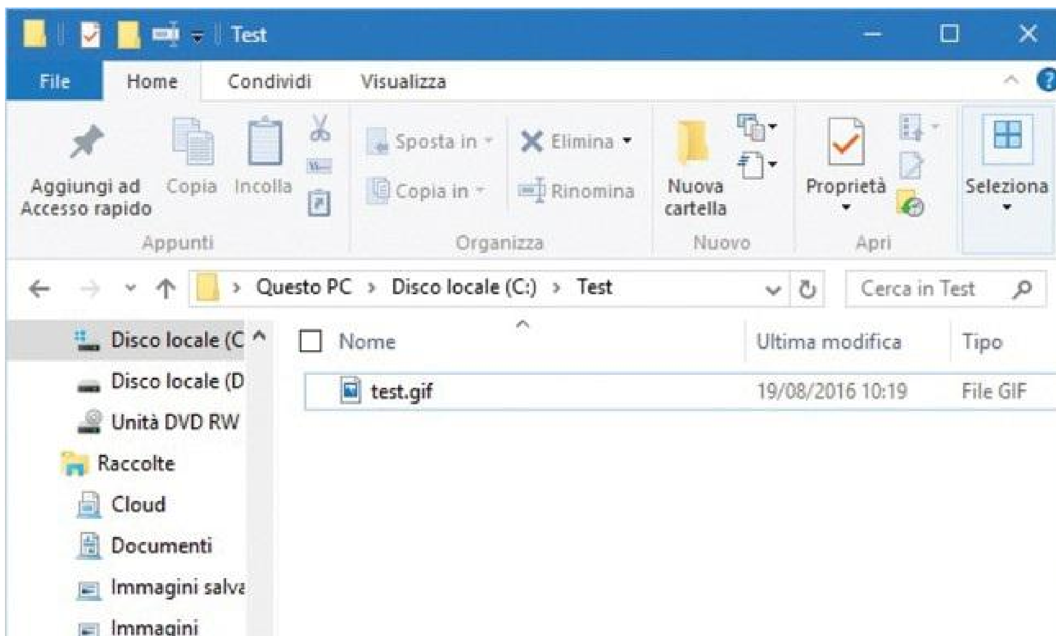
dimensioni:



VISUALIZZARE IMMAGINI

Per visualizzare un'immagine su una tela con tkinter, come prima cosa si deve caricare l'immagine e poi usare la funzione `create_image` sull'oggetto `canvas`.

Qualsiasi immagine si carichi deve essere in una directory accessibile a Python. Per questo esempio, abbiamo messo la nostra immagine *test.gif* nella directory C:\, che è la directory radice (quella fondamentale) dell'unità disco C:, ma voi potete metterla dove volete.



Se usate un sistema Mac o Linux, potete mettere l'immagine nella directory *Home*. Se non riuscite a salvare dei file nell'unità C:, potete mettere l'immagine sul desktop.

NOTA

Con `tkinter`*, si possono caricare solo immagini GIF, cioè file di immagine con estensione* .gif*. Potete visualizzare anche altri tipi di immagini, per esempio PNG (*.png*) e JPG (*.jpg*), ma dovrete usare un altro modulo, per esempio la Python Imaging Library (*http://www.pythonware.com/products/pil/*).*

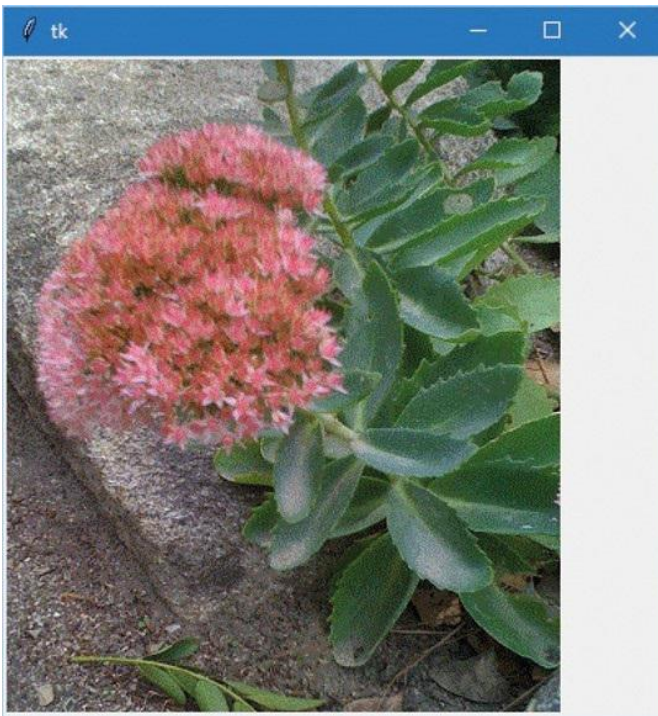
Possiamo visualizzare l'immagine *test.gif* in questo modo:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
mia_immagine = PhotoImage(file='c:\\test.gif')
canvas.create_image(0, 0, anchor=NW, image=mia_immagine)
```

Nelle prime quattro righe, impostiamo la tela come negli esempi precedenti. Nella quinta riga, l'immagine viene caricata nella variabile `mia_immagine`. Creiamo `PhotoImage` con la directory `'c:\\test.gif'`. Se avete salvato la PhotoImage sul desktop, dovete creare la `PhotoImage` indicando quella posizione, che sarà qualcosa di simile a questo:

```
mia_immagine = PhotoImage(file='C:\\Users\\Giovanni Rossi\\
Desktop\\test.gif')
```

Una volta che l'immagine è stata caricata nella variabile, `canvas.create_image(0, 0, anchor=NW, image=mia_immagine)` la visualizza con la funzione `create_image`. Le coordinate (0, 0) sono quelle del punto a partire dal quale verrà visualizzata l'immagine e `anchor=NW` dice alla funzione di usare l'angolo superiore sinistro (`NW` sta per "nord-ovest") dell'immagine come punto di partenza nel disegnarla (altrimenti userebbe il centro dell'immagine come punto di partenza, per impostazione predefinita). L'ultimo parametro per nome, `image`, punta alla variabile per l'immagine caricata. Ecco il risultato:



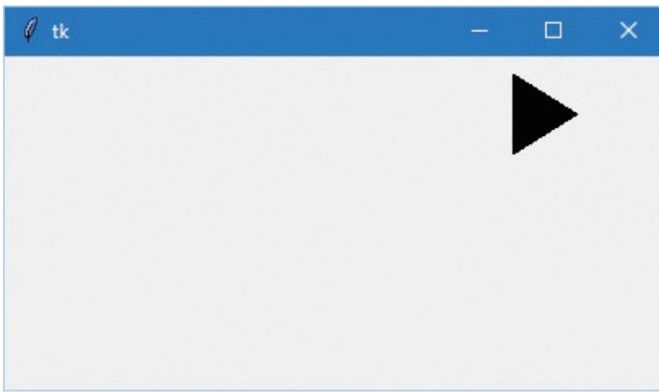
CREARE UN’ANIMAZIONE DI BASE

Abbiamo visto fin qui come creare disegni statici, cioè immagini che non si muovono. E se provassimo a creare un’animazione?

L’animazione non è necessariamente una specialità del modulo `tkinter`, ma può comunque essere utile per qualcosa di semplice. Per esempio, possiamo creare un triangolo pieno e poi farlo muovere lungo lo schermo con il codice seguente (non dimenticate: selezionate **File** > **New File**, salvate il lavoro, poi eseguite il codice con **Run** > **Run Module**):

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=200)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
        canvas.move(1, 5, 0)
        tk.update()
        time.sleep(0.05)
```

Quando eseguite questo codice, il triangolo comincerà a muoversi sullo schermo, fino alla fine del suo percorso:



Come funziona? Come prima, nelle prime tre righe abbiamo importato `tkinter` e visualizzato un canvas. Nella quarta riga, abbiamo creato il triangolo con questa funzione:

```
canvas.create_polygon(10, 10, 10, 60, 50, 35)
```

NOTA

Quando inserite questa riga, sullo schermo verrà stampato un numero: è un identificatore del poligono. Lo si può usare per fare riferimento alla figura, come vedremo nell'esempio seguente.

Poi abbiamo creato un ciclo `for` che conta da 0 a 59, partendo da `for x in range(0, 60)`:. Il blocco di codice nel ciclo fa spostare il triangolo. La funzione `canvas.move` sposta qualsiasi oggetto sia stato disegnato aggiungendo dei valori alle sue coordinate x e y. Per esempio, con `canvas.move(1, 5, 0)` spostiamo l'oggetto con ID 1 (l'identificatore del triangolo) di 5 pixel verso destra e di 0 pixel verso il basso. Per riportarlo alla posizione precedente, potremmo usare la funzione con un valore negativo, `canvas.move(1, -5, 0)`.

La funzione `tk.update()` costringe `tkinter` ad aggiornare lo schermo (ridisegnarlo). Se non avessimo usato `update`, `tkinter` avrebbe aspettato la conclusione del ciclo prima di spostare il triangolo, il che significa che l'avreste visto saltare alla posizione finale, anziché spostarsi progressivamente. La riga finale del ciclo, `time.sleep(0.05)`, dice a Python di aspettare un ventesimo di secondo (0,05 secondi), prima di procedere.

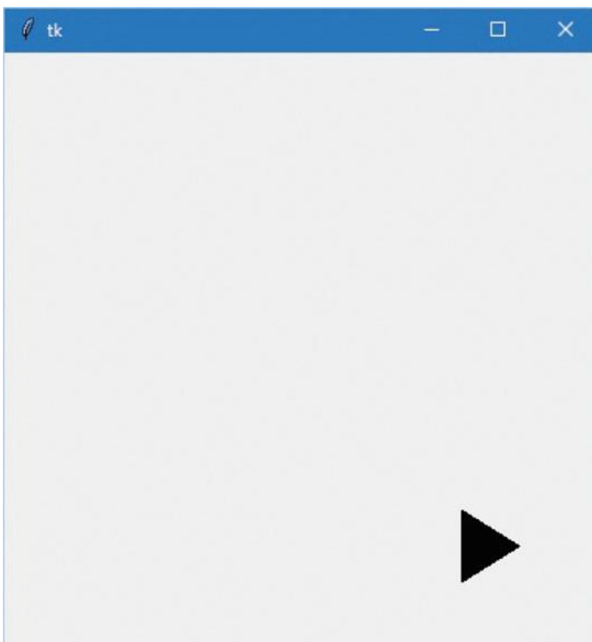
Per far spostare il triangolo diagonalmente sullo schermo, possiamo modificare il codice chiamando `move(1, 5, 5)`. Create un nuovo file (**File** > **New File**) per questo codice:

```
import time
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
for x in range(0, 60):
        canvas.move(1, 5, 5)
        tk.update()
        time.sleep(0.05)
```

Questo codice è diverso dal precedente per due aspetti:

- Abbiamo creato una tela di altezza 400, anziché 200 pixel, con `canvas = Canvas(tk, width=400, height=400)`.
- Sommiamo 5 alle coordinate x e y del triangolo con `canvas.move(1, 5, 5)`.

Una volta salvato il codice, lo potete eseguire, ed ecco la posizione del triangolo alla fine del ciclo:



Per far spostare il triangolo diagonalmente da questa posizione a quella di partenza, usate gli incrementi negativi `-5, -5`. Aggiungete questo codice alla fine del file:

```
for x in range(0, 60):
        canvas.move(1, -5, -5)
        tk.update()
        time.sleep(0.05)
```

FAR REAGIRE UN OGGETTO

Possiamo fare in modo che il triangolo reagisca quando qualcuno preme un tasto, utilizzando gli *event binding* (collegamenti a eventi). Gli eventi sono cose che si verificano mentre un programma è in esecuzione: qualcuno muove il mouse, preme

un tasto o chiude una finestra. Si può dire a `tkinter` di fare attenzione a questi eventi e, quando se ne verifica uno, di rispondere in qualche modo.

Per cominciare a gestire gli eventi (cioè per far fare qualcosa a Python quando si verifica un evento), creiamo prima una funzione. La parte di collegamento (*binding*) arriva quando si dice a `tkinter` che una particolare funzione è legata (o associata) a un evento specifico; in altre parole, verrà chiamata automaticamente da `tkinter` per gestire quell'evento.

Per esempio, per fare sì che il triangolo si muova quando si preme il tasto INVIO, possiamo definire questa funzione:

```
def movetriangle(event):
        canvas.move(1, 5, 0)
```

La funzione prende un unico parametro (`event`), che `tkinter` usa per inviare informazioni alla funzione in merito all'evento. Ora diciamo a `tkinter` che questa funzione deve essere usata per un particolare evento, utilizzando la funzione `bind_all` sul canvas. Il codice completo è questo:

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
    canvas.move(1, 5, 0)
canvas.bind_all('<KeyPress-Return>', movetriangle)
```

Il primo parametro di questa funzione descrive l'evento a cui vogliamo che `tkinter` faccia attenzione. In questo caso, si chiama `<KeyPress-Return>`, ovvero "pressione del tasto INVIO O RETURN". Diciamo a `tkinter` che la funzione `movetriangle` deve essere chiamata ogni volta che si verifica questo evento `KeyPress`. Eseguite questo codice, fate clic sul canvas con il mouse, poi provate a premere INVIO.

E se volessimo cambiare la direzione del triangolo in base ai tasti premuti, per esempio i tasti freccia? Non è difficile. Basta modificare la funzione `movetriangle` in questo modo:

```
def movetriangle(event):
    if event.keysym == 'Up':
        canvas.move(1, 0, -3)
    elif event.keysym == 'Down':
        canvas.move(1, 0, 3)
    elif event.keysym == 'Left':
        canvas.move(1, -3, 0)
    else:
        canvas.move(1, 3, 0)
```

L'oggetto `event` passato a `movetriangle` contiene diverse variabili. Una è `keysym` (abbreviazione di *key symbol*, simbolo del tasto), una stringa contenente il valore del tasto premuto (`Up` è la freccia verso l'alto, `Down` quella verso il basso, `Left` e `Right` sinistra e destra). La riga `if event.keysym == 'Up'`: dice che se la variabile `keysym` contiene la stringa `'Up'` dobbiamo chiamare `canvas.move` con i parametri `(1, 0, -3)`, come facciamo nella riga successiva. Se `keysym` contiene `'Down'`, la condizione `elif event.keysym == 'Down'`: sarà vera e verrà chiamata la funzione con i parametri `(1, 0, 3)`, e così via.

Ricordate: il primo parametro è il numero identificativo della forma disegnata, il secondo è il valore da sommare alla coordinata x, il terzo il valore da sommare alla coordinata y.

Poi diciamo a `tkinter` di usare la funzione `movetriangle` per gestire gli eventi di quattro diversi tasti (up, down, left e right). Qui di seguito si vede come si presenta il codice a questo punto. Per inserire questo codice, sarà tutto più facile se create un nuovo file (selezionando **File > New File**). Prima di eseguire il codice, salvate il file con un nome significativo, per esempio *triangoloinmoto.py*.

```
from tkinter import *
tk = Tk()
canvas = Canvas(tk, width=400, height=400)
canvas.pack()
canvas.create_polygon(10, 10, 10, 60, 50, 35)
def movetriangle(event):
❶   if event.keysym == 'Up':
❷       canvas.move(1, 0, -3)
❸   elif event.keysym == 'Down':
❹       canvas.move(1, 0, 3)
❺   elif event.keysym == 'Left':
❻       canvas.move(1, -3, 0)
❼   else:
❽       canvas.move(1, 3, 0)
canvas.bind_all('<KeyPress-Up>', movetriangle)
canvas.bind_all('<KeyPress-Down>', movetriangle)
canvas.bind_all('<KeyPress-Left>', movetriangle)
canvas.bind_all('<KeyPress-Right>', movetriangle)
```

Nella prima riga della funzione `movetriangle`, controlliamo se `keysym` contiene `'Up'`, in ❶. Se sì, spostiamo il triangolo verso l'alto con la funzione `move` e i parametri `1, 0, -3`, in ❷. Il primo è l'identificatore del triangolo, il secondo lo spostamento verso destra (non vogliamo spostarlo in orizzontale, perciò è 0) e il terzo è lo spostamento verso il basso (negativo, perché vogliamo andare verso l'alto, -3 pixel).

Poi verifichiamo se `keysym` contiene `'Down'` in ❸ e, se sì, spostiamo il triangolo

verso il basso (di 3 pixel) in ❹. L'ultimo controllo è se il valore è `'Left'`, in ❺ e, se sì, spostiamo il triangolo verso sinistra (-3 pixel) in ❻. Se la variabile non contiene nessuno di questi valori, l'ultimo `else` in ❼ sposta il triangolo verso destra in ❽. Ora il triangolo si muoverà nella direzione indicata dal tasto freccia che viene premuto.

ALTRI MODI PER USARE L'IDENTIFICATORE

Ogni volta che si usa una funzione `create_` del canvas, come `create_polygon` o `create_rectangle`, viene restituito un identificatore. Questo numero identificativo può essere utilizzato con altre funzioni, come abbiamo fatto in precedenza con la funzione `move`:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> canvas.create_polygon(10, 10, 10, 60, 50, 35)
1
>>> canvas.move(1, 5, 0)
```

Il problema è che `create_polygon` non restituisce sempre 1. Per esempio, se avete creato altre forme, potrebbe restituire 2, 3 o anche 100 (a seconda del numero delle forme che avete creato). Se però modifichiamo il codice in modo da memorizzare in una variabile il valore restituito, e poi usiamo la variabile (anziché fare riferimento semplicemente al numero 1), il codice funzionerà, indipendentemente da quale sia il numero restituito:

```
>>> miotriangolo = canvas.create_polygon(10, 10, 10, 60, 50, 35)
>>> canvas.move(miotriangolo, 5, 0)
```

La funzione `move` ci permette di spostare oggetti sullo schermo mediante il loro identificatore. Esistono però altre funzioni che possono modificare in qualche modo quel che abbiamo disegnato. Per esempio, la funzione `itemconfig` del canvas si può usare per modificare qualcuno dei parametri di una forma, per esempio i colori di riempimento e di contorno.

Supponiamo di aver creato un triangolo rosso:

```
>>> from tkinter import *
>>> tk = Tk()
>>> canvas = Canvas(tk, width=400, height=400)
>>> canvas.pack()
>>> miotriangolo = canvas.create_polygon(10, 10, 10, 60, 50, 35,
fill='red')
```

Possiamo cambiare il colore del triangolo utilizzando `itemconfig`, con l'identificare come primo parametro. Il codice che segue dice: "Cambia in blu il colore di riempimento dell'oggetto identificato dal numero conservato nella

variabile `miotriangolo`”:

```
>>> canvas.itemconfig(miotriangolo, fill='blue')
```

Possiamo anche dare un colore diverso al contorno del triangolo, sempre con l’identificatore come primo parametro:

```
>>> canvas.itemconfig(miotriangolo, outline='red')
```

Più avanti, vedremo come apportare ulteriori modifiche a un disegno, per esempio nasconderlo e poi renderlo nuovamente visibile. Questo ci sarà utile, quando inizieremo a scrivere giochi, nel prossimo capitolo.

CHE COSA AVETE IMPARATO

In questo capitolo, avete usato il modulo `tkinter` per disegnare semplici forme geometriche, visualizzare immagini e semplici animazioni. Avete visto come usare gli *event binding* per far sì che i disegni reagiscano alla pressione dei tasti, il che sarà utile quando inizieremo a programmare un gioco. Avete visto come le funzioni di creazione di forme in `tkinter` restituiscano un numero identificativo, utilizzabile per modificare le forme dopo che sono state disegnate, per esempio per spostarle sullo schermo o per modificarne il colore.

ROMPICAPO DI PROGRAMMAZIONE

Cimentatevi con questi esercizi, per sperimentare ancora con il modulo `tkinter` e con qualche semplice animazione. Le soluzioni si trovano sul sito *http://python-for-kids.com/*.

1. RIEMPIRE LO SCHERMO DI TRIANGOLI

Create un programma che usi `tkinter` per riempire lo schermo di triangoli. Poi modificare il codice così da riempire lo schermo con triangoli di colori diversi.

2. IL TRIANGOLO MOBILE

Modificate il codice del triangolo che si muove (“Creare animazioni semplici” a pagina 183) perché si sposti sullo schermo prima a destra, poi in basso, poi a sinistra e infine verso l’alto fino alla posizione di partenza.

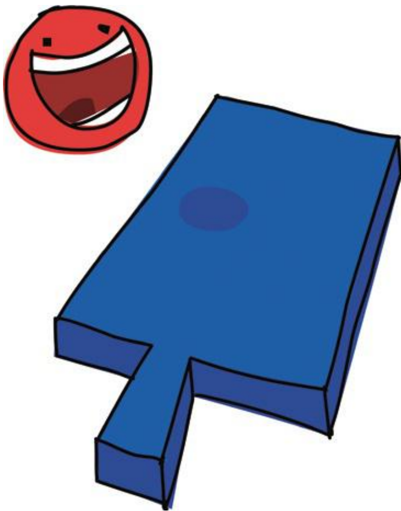
3. LA FOTO CHE SI MUOVE

Provate a visualizzare con `tkinter` una foto. Ricordate: deve essere un’immagine GIF! Riuscite a spostarla sullo schermo?

AGGIUNGERE LA RACCHETTA

Non c'è molto di che divertirsi con una palla che rimbalza, se non si ha nulla con cui colpirla. È venuto il momento di creare una racchetta!

Cominciate aggiungendo il codice seguente subito dopo la classe `Ball`, per creare una racchetta (la aggiungerete creando una nuova riga sotto la funzione `draw` della palla):



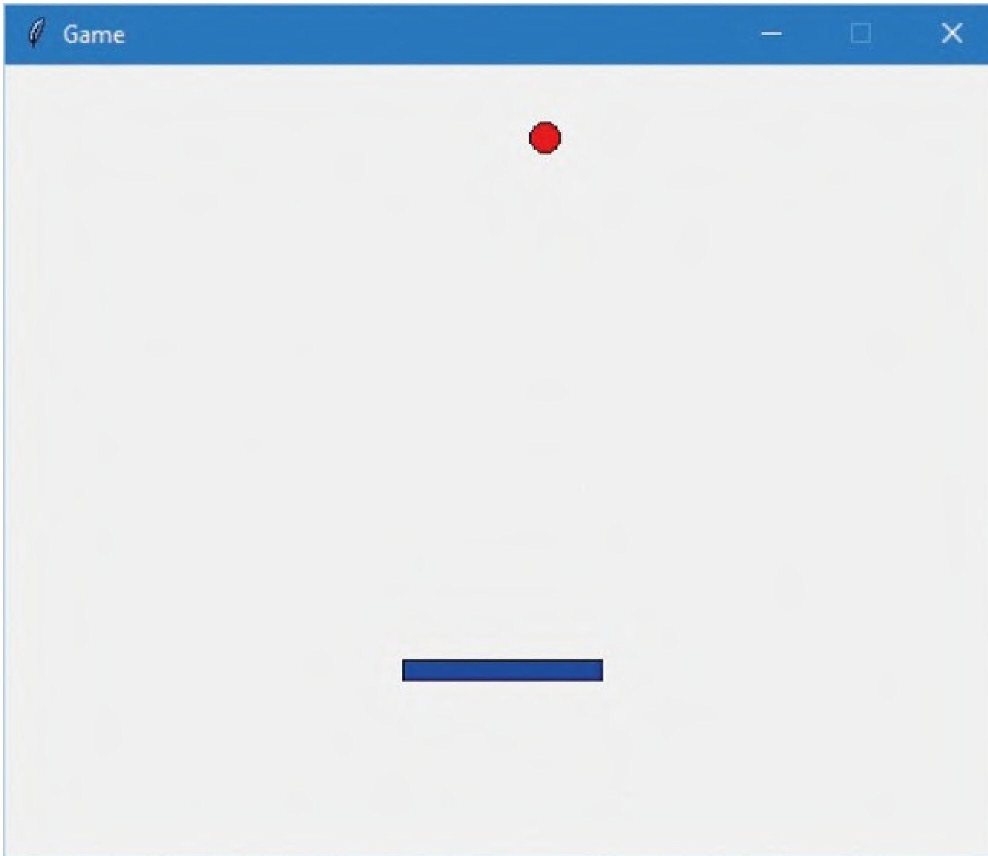
```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

class Paddle:
    def __init__(self, canvas, color):
        self.canvas = canvas
        self.id = canvas.create_rectangle(0, 0, 100, 10, \
                fill=color)
        self.canvas.move(self.id, 200, 300)

    def draw(self):
        pass
```

Il codice aggiunto è quasi identico a quello della classe `Ball`, tranne che chiamiamo la funzione `create_rectangle` (anziché `create_oval`) e spostiamo il rettangolo nella posizione 200, 300 (200 pixel da sinistra e 300 pixel dall'alto).

Poi, alla fine del listato, create un oggetto della classe `Paddle` e modificate il ciclo principale in modo che chiami la funzione `draw` della racchetta, così:

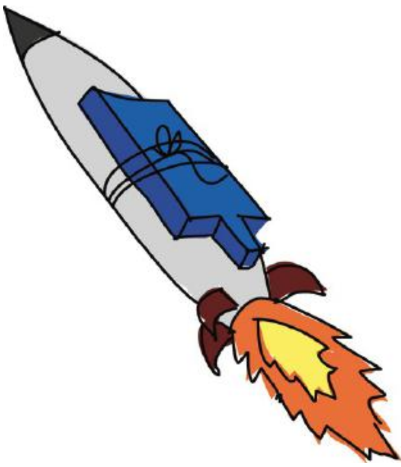
```
paddle = Paddle (canvas, 'blue')
ball = Ball (canvas, 'red')
while 1:
    ball.draw()
    paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Se eseguite il gioco a questo punto, dovreste vedere la palla che rimbalza e una racchetta rettangolare immobile:



SPOSTARE LA RACCHETTA

Per spostare la racchetta a sinistra e a destra, useremo i collegamenti a eventi per collegare i tasti con le frecce sinistra e destra a nuove funzioni della classe `Paddle`. Quando il giocatore preme il tasto freccia sinistra, la variabile `x` verrà impostata a `-2` (per spostare a sinistra); la pressione del tasto freccia destra imposta invece la variabile `x` a `2` (per spostare a destra).



Il primo passo è aggiungere la variabile d'oggetto x alla funzione __init__ della classe Paddle, e una variabile per la larghezza del canvas, come abbiamo fatto per la classe Ball:

```
def __init__(self, canvas, color):
    self.canvas = canvas
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
    self.canvas.move(self.id, 200, 300)
    self.x = 0
    self.canvas_width = self.canvas.winfo_width()
```

Ora abbiamo bisogno delle funzioni per cambiare direzione, verso sinistra (turn_left) e verso destra (turn_right). Le aggiungiamo subito dopo la funzione draw:

```
def turn_left(self, evt):
    self.x = -2

def turn_right(self, evt):
    self.x = 2
```

Possiamo legare queste funzioni al tasto corrispondente nella funzione __init__ della classe. Abbiamo usato il collegamento (*binding*) in "Far reagire un oggetto" a pagina 186, in modo che Python chiamasse una funzione alla pressione di un tasto. In questo caso, leghiamo la funzione turn-left della nostra classe Paddle al tasto freccia sinistra con il nome di evento '<KeyPress-Left>'. Poi leghiamo la funzione turn-right al tasto freccia destra usando il nome di evento '<KeyPress-Right>'. La funzione __init__ ora è fatta in questo modo:

```
def __init__(self, canvas, color):
    self.canvas = canvas
    self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
    self.canvas.move(self.id, 200, 300)
    self.x = 0
    self.canvas_width = self.canvas.winfo_width()
    self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
    self.canvas.bind_all('<KeyPress-Right>', self.turn_right)
```

La funzione draw per la classe Paddle è simile a quella per la classe Ball:

```
def draw(self):
    self.canvas.move(self.id, self.x, 0)
    pos = self.canvas.coords(self.id)
    if pos[0] <= 0:
        self.x = 0
    elif pos[2] >= self.canvas_width:
        self.x = 0
```

Usiamo la funzione `move` del canvas per spostare la racchetta nella direzione della variabile `x` con `self.canvas. move(self.id, self.x, 0)`. Poi ricaviamo le coordinate della racchetta per vedere se ha raggiunto il bordo sinistro o destro dello schermo utilizzando il valore in `pos`.

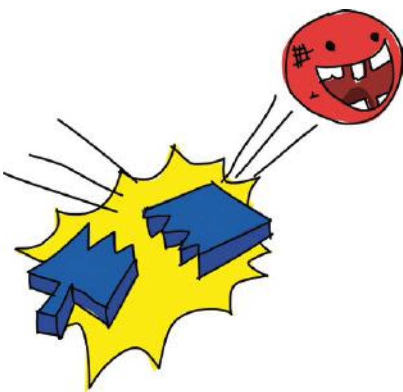
Anziché rimbalzare come la palla, la racchetta deve smettere di muoversi. Così, quando la coordinata x di sinistra (`pos[0]`) è minore o uguale a 0 (`<=0`), impostiamo la variabile `x` a 0 con `self.x = 0`. Nello stesso modo, quando la coordinata x di destra (`pos[2]`) è maggiore o uguale alla larghezza del canvas (`>= self.canvas_width`), impostiamo la variabile `x` a 0 con `self-x = 0`.

NOTA

Se eseguite il programma ora, dovrete fare clic sul canvas perché il gioco riconosca le azioni dei tasti freccia. Con il clic il canvas riceve il focus, cioè “sa” di avere la responsabilità di reagire quando qualcuno preme un tasto sulla tastiera.

SCOPRIRE QUANDO LA PALLA COLPISCE LA RACCHETTA

A questo punto del nostro codice, la palla non colpirà la racchetta, ma vi passerà attraverso come se fosse un fantasma. La palla deve sapere quando colpisce la racchetta, esattamente come deve sapere quando colpisce uno dei bordi.



Possiamo risolvere il problema aggiungendo del codice alla funzione `draw` (dove abbiamo già il codice che controlla l’arrivo ai bordi), ma è meglio spostare questo tipo di codice in nuove funzioni, per suddividere il tutto in parti più piccole. Se mettiamo troppo codice in uno stesso luogo (all’interno di una funzione, per esempio), quel codice diventa più difficile da capire. Facciamo i cambiamenti necessari.

Per prima cosa, modifichiamo la funzione `__init__` della palla, per poter passare

l'oggetto `paddle` come parametro:

```
class Ball:
❶    def __init__(self, canvas, paddle, color):
        self.canvas = canvas
❷        self.paddle = paddle
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
```

Notate che in ❶ abbiamo modificato il parametro di `__init__` in modo da includere la racchetta. Poi, in ❷, assegniamo il parametro `paddle` alla variabile di oggetto paddle.

Avendo salvato l'oggetto `paddle`, dobbiamo modificare il codice con cui creiamo l'oggetto `ball`. Questo cambiamento va in fondo al programma, subito prima del ciclo principale:

```
paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while 1:
    ball.draw()
    paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

Il codice di cui abbiamo bisogno per vedere se la palla ha colpito la racchetta è un po' più complicato di quello con cui si controlla se sono stati raggiunti i bordi. Chiameremo questa funzione `hit_paddle` e la aggiungeremo alla funzione `draw` della classe `Ball`, dove vediamo se la palla ha raggiunto il fondo dello schermo:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if pos[3] >= self.canvas_height:
            self.y = -3
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3
```

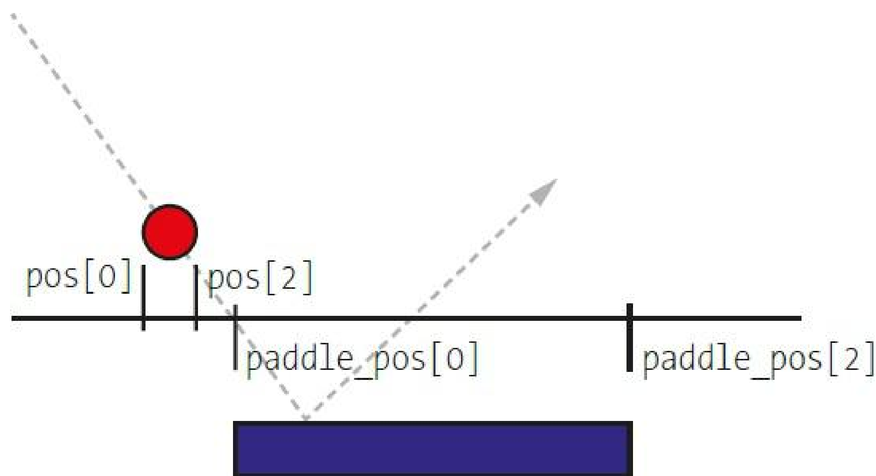
Come potete vedere nel nuovo codice che abbiamo aggiunto, se `hit_paddle` restituisce `True`, cambiamo la direzione della palla impostando la variabile `y` a `-3` con `self.y = -3`. Non provate a eseguire il gioco a questo punto: non abbiamo ancora creato la funzione `hit_paddle`. Facciamolo ora.

Aggiungete la funzione `hit_paddle` subito prima della funzione `draw`.

```
❶    def hit_paddle(self, pos):
❷        paddle_pos = self.canvas.coords(self.paddle.id)
❸        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
❹            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                 return True
         return False
```

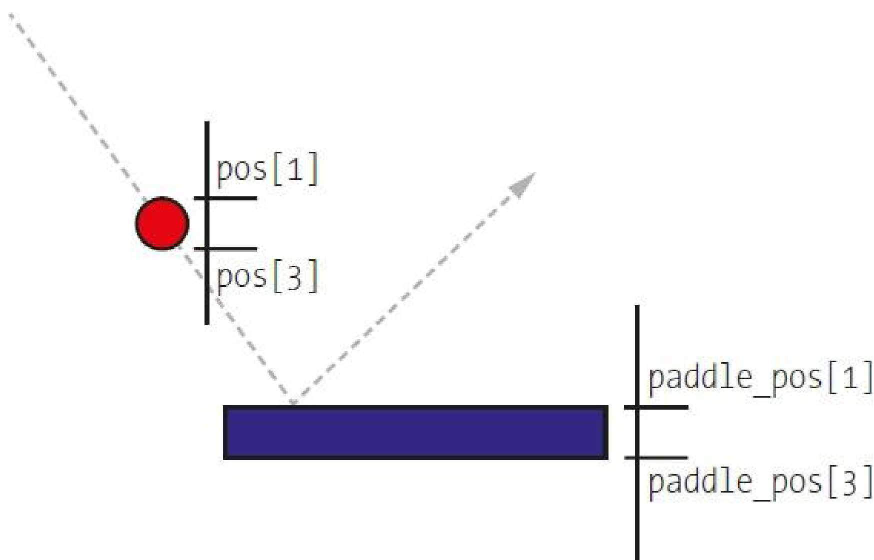
Come prima cosa, definiamo la funzione con il parametro `pos`, in ❶. Questa riga contiene le coordinate attuali della palla. Poi, in ❷, otteniamo le coordinate della racchetta e le conserviamo nella variabile `paddle_pos`.

In ❸, abbiamo la prima parte del nostro enunciato `if-then`, e diciamo "Se il lato destro della palla è maggiore del lato sinistro della racchetta, e il lato sinistro della palla è minore del lato destro della racchetta…". Qui, `pos[2]` contiene la coordinata `x` per l'estremo destro della palla, `pos[0]` contiene la coordinata `x` del suo estremo sinistro.
La variabile `paddle_pos[0]` contiene la coordinata `x` dell'estremo sinistro della racchetta, `paddle_pos[2]` contiene la coordinata x del suo estremo destro. il diagramma seguente mostra queste coordinate quando la palla sta per colpire la racchetta.



La palla sta scendendo verso la racchetta, ma si vede che l'estremo destro della palla (`pos[2]`) non ha ancora superato l'estremo sinistro della racchetta (che è `paddle_pos[0]`).

In ❹, vediamo se l'estremo inferiore della palla (`pos[3]`) è fra la parte superiore della racchetta (`paddle_pos[1]`) e la sua parte inferiore (`paddlepos[3]`). Nel diagramma seguente, potete vedere che l'estremo inferiore della palla (`pos[3]`) non ha ancora raggiunto la parte superiore della racchetta (`paddle_pos[1]`).



Perciò, in base alla posizione attuale della palla, la funzione `hit_paddle` restituirebbe False.

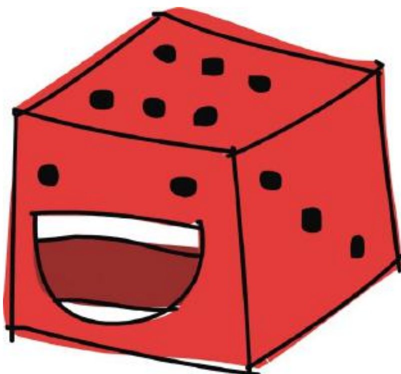
NOTA

Perché dobbiamo vedere se il punto inferiore della palla si trova fra la parte superiore e quella inferiore della racchetta? Perché ogni volta che spostiamo la palla sullo schermo, la facciamo spostare di 3 pixel alla volta. Se controllassimo semplicemente se la palla ha raggiunto la parte superiore della racchetta (`pos[1]`*), potremmo già aver superato quella posizione. In quel caso la palla continuerebbe lungo la sua traiettoria e passerebbe attraverso la racchetta senza fermarsi.*

AGGIUNGERE UN ELEMENTO DI CASUALITÀ

È venuto il momento di trasformare il nostro programma in un gioco, anziché in un semplice movimento di palla e racchetta. In un gioco deve esserci un elemento di casualità, un qualche modo per cui il giocatore può perdere. Nel nostro gioco, così com'è, la palla rimbalzerà per sempre, perciò non c'è niente da perdere.

Concluderemo il nostro gioco aggiungendo del codice che dice che il gioco si conclude se la palla colpisce il fondo del canvas (in altre parole, se cade a terra).



Come prima cosa, aggiungiamo la variabile di oggetto `hit_bottom` in fondo alla

funzione `__init__` della classe `Ball`:

```
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False
```

Poi modifichiamo il ciclo principale, in fondo al programma, in questo modo:

```
while 1:
    if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

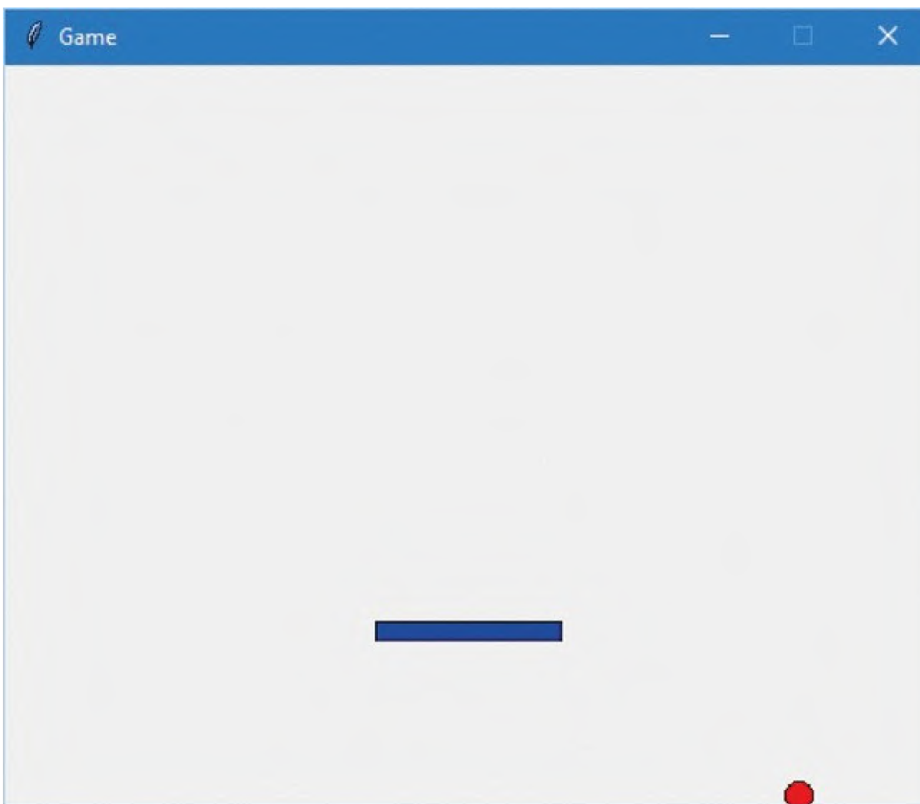
Ora il ciclo continua a verificare `hit_bottom`, per vedere se la palla ha colpito il fondo dello schermo. Il codice continuerà a far spostare palla e racchetta solo se la palla non ha colpito il fondo, come si vede nell'enunciato `if`. Il gioco termina quando palla e racchetta smettono di muoversi.

L'ultima modifica è nella funzione `draw` della classe `Ball`:

```
    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if self.hit_paddle(pos) == True:
        self.y = -3
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3
```

Abbiamo modificato l'enunciato `if` in modo che controlli se la palla ha colpito il fondo dello schermo (cioè se la sua posizione è maggiore o uguale a `canvas_height`). Se è così, nella riga successiva, impostiamo `hit_bottom` a `True`, anziché modificare il valore della variabile `y`, perché non c'è più bisogno di far rimbalzare la palla, una volta che ha colpito il fondo dello schermo.

Se adesso eseguite il gioco e non riuscite a colpire la palla con la racchetta, ogni movimento sullo schermo dovrebbe cessare, e il gioco finirà non appena la palla tocca il fondo del canvas:



Il programma completo ora sarà come nel codice seguente. Se avete difficoltà a far funzionare il gioco, confrontate quello che avete inserito con questo codice.

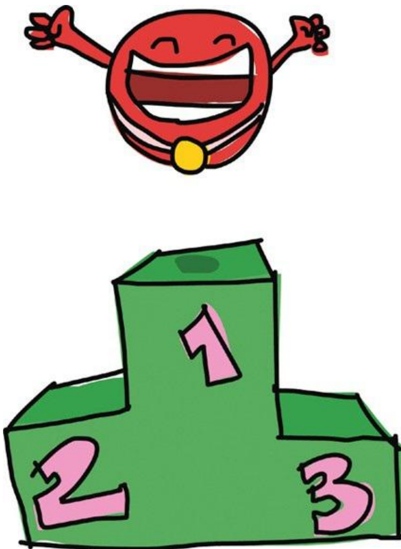
```
from tkinter import *
import random
import time

tk = Tk()
tk.title("Game")
tk.resizable(0, 0)
tk.wm_attributes("-topmost", 1)
canvas = Canvas(tk, width=500, height=400, bd=0,
highlightthickness=0)
canvas.pack()
tk.update()

class Ball:
    def __init__(self, canvas, paddle, color):
        self.canvas = canvas
        self.paddle = paddle
        self.id = canvas.create_oval(10, 10, 25, 25, fill=color)
        self.canvas.move(self.id, 245, 100)
```

```
        starts = [-3, -2, -1, 1, 2, 3]
        random.shuffle(starts)
        self.x = starts[0]
        self.y = -3
        self.canvas_height = self.canvas.winfo_height()
        self.canvas_width = self.canvas.winfo_width()
        self.hit_bottom = False

    def draw(self):
        self.canvas.move(self.id, self.x, self.y)
        pos = self.canvas.coords(self.id)
        if pos[1] <= 0:
            self.y = 3
        if self.hit_paddle(pos) == True:
            self.y = -3
        if pos[3] >= self.canvas_height:
            self.hit_bottom = True
        if pos[0] <= 0:
            self.x = 3
        if pos[2] >= self.canvas_width:
            self.x = -3

    def hit_paddle(self, pos):
        paddle_pos = self.canvas.coords(self.paddle.id)
        if pos[2] >= paddle_pos[0] and pos[0] <= paddle_pos[2]:
            if pos[3] >= paddle_pos[1] and pos[3] <= paddle_pos[3]:
                return True
        return False

    class Paddle:
        def __init__(self, canvas, color):
            self.canvas = canvas
            self.id = canvas.create_rectangle(0, 0, 100, 10, fill=color)
            self.canvas.move(self.id, 200, 300)
            self.x = 0
            self.canvas_width = self.canvas.winfo_width()
            self.canvas.bind_all('<KeyPress-Left>', self.turn_left)
            self.canvas.bind_all('<KeyPress-Right>', self.turn_right)

        def turn_left(self, evt):
            self.x = -2

        def turn_right(self, evt):
            self.x = 2

        def draw(self):
            self.canvas.move(self.id, self.x, 0)
            pos = self.canvas.coords(self.id)
            if pos[0] <= 0:
                self.x = 0
            elif pos[2] >= self.canvas_width:
                self.x = 0

paddle = Paddle(canvas, 'blue')
ball = Ball(canvas, paddle, 'red')

while 1:
    if ball.hit_bottom == False:
        ball.draw()
        paddle.draw()
    tk.update_idletasks()
    tk.update()
    time.sleep(0.01)
```

CHE COSA AVETE IMPARATO

In questo capitolo, abbiamo finito di creare il nostro primo gioco con il modulo `tkinter`. Abbiamo creato classi per la racchetta utilizzata nel gioco e abbiamo usato le coordinate per verificare quando la palla colpisce la racchetta o i bordi del canvas di gioco. Abbiamo usato i collegamenti a eventi per legare i tasti freccia sinistra e destra al movimento della racchetta e abbiamo usato il ciclo principale per

chiamare la funzione draw e animare la racchetta. Infine, abbiamo modificato il codice per inserire nel gioco un elemento di casualità e ora il gioco si conclude quando la palla colpisce il fondo del canvas.



ROMPICAPO DI PROGRAMMAZIONE

Per il momento, il nostro gioco è molto semplice. Potete modificare molte cose, per creare un gioco più professionale. Provate ad arricchire il codice secondo i suggerimenti che seguono, poi verificate le vostre soluzioni sul sito *http://python-for-kids.com/*.

1. RITARDARE L'AVVIO DEL GIOCO

Il nostro gioco inizia un po' troppo rapidamente, e bisogna fare clic sul canvas perché riconosca la pressione dei tasti freccia sulla tastiera. Potete aggiungere un ritardo all'avvio del gioco, in modo da dare al giocatore un tempo sufficiente per fare clic sul canvas? Ancora meglio, riuscite ad aggiungere un collegamento a evento per un clic del mouse, in modo che il gioco parta solo in quel momento?

Suggerimento 1: avete già aggiunto collegamenti a eventi per la classe `Paddle`: quello può essere un punto di partenza.

Suggerimento 2: il collegamento a evento per il pulsante sinistro del mouse è la stringa `'<Button-1>'`.

2. UN ADEGUATO "GAME OVER"

Quando il gioco finisce, semplicemente tutto si congela, e non è un modo molto amichevole nei confronti del giocatore. Provate ad aggiungere il testo "Game Over" quando la palla colpisce il fondo dello schermo. Potete usare la funzione `create_text`, ma potreste trovare utile anche il parametro con nome `state` (prende valori come `normal` e `hidden`, cioè "normale" e "nascosto"). Date un'occhiata a `itemconfig` in "Altri modi per usare l'identificatore" a pagina 188. Come ulteriore sfida, aggiungete un ritardo, perché il testo non appaia subito.

3. ACCELERARE LA PALLA

Se giocate a tennis, sapete che, quando una palla colpisce la racchetta, a volte si allontana a una velocità superiore a quella con cui è arrivata, a seconda della forza del vostro colpo. Nel nostro gioco la palla viaggia sempre alla stessa velocità, qualunque cosa faccia la racchetta. Provate a modificare il programma, in modo che la velocità della racchetta sia trasferita alla velocità della palla.

4. REGISTRARE IL PUNTEGGIO

Che ne dite di registrare il punteggio? Ogni volta che la palla colpisce la racchetta, il punteggio deve aumentare. Provate a visualizzare il punteggio nell'angolo superiore destro del canvas. Potete riguardare come si usa `itemconfig`, a pagina 188, per ricavarne un suggerimento.