Una calcolatrice programmabile

In questo capitolo

- Il primo programma: "Hello, World!"
- Personalizzare IDLE
- Valutazione di un'espressione
- Operazioni aritmetiche ed espressioni
- Variabili e operatore di assegnamento
- Le funzioni predefinite
- I tipi di dato
- La funzione print()
- Per qualche calcolo in più…
- Stile di programmazione
- Help!
- Proposte di variazione sul tema
- Cosa hai imparato

3

In questo capitolo vedrai alcune funzionalità dell'ambiente di IDLE. Interagendo direttamente con l'interprete, imparerai alcune istruzioni base di Python, iniziando a fare pratica con la sua capacità di compiere calcoli con operatori semplici e composti. Infine, grazie all'uso delle variabili per memorizzare e recuperare valori e alle funzioni predefinite, scriverai alcuni programmi per risolvere semplici problemi di matematica finanziaria e di fisica.

Il primo programma: "Hello, World!"

In IDLE, per scrivere il primo programma, che visualizza semplicemente un saluto, seleziona *New File* dal menu *File*, copia il codice riportato nel listato seguente e salva in una cartella del tuo PC il programma come **hello_world.py** (i file di Python hanno estensione **.py**), selezionando *Save As* dal menu *File* o premendo la combinazione di tasti CTRL+S.

NOTA

Una *stringa* è una sequenza di caratteri delimitata da virgolette (") o apici ('). I caratteri sono: le lettere dell'alfabeto latino, minuscole e maiuscole, le cifre arabe, i simboli di punteggiatura (la virgola, i due punti,...), lo spazio e i *caratteri di controllo* (per esempio il carattere per andare a capo). Una stringa, che si estende su più righe (*multilinea*), è delimitata da tre virgolette consecutive (""") oppure tre apici (''').

hello_world.py

```
"""Il nostro primo programma in Python"""
print("Hello, World!")    # "Stampa" a video un saluto al mondo
```

Esegui il programma selezionando *Run Module* dal menu *Run* oppure premendo il tasto F5. Adesso ci sono due finestre aperte (Figura 3.1): in una si trova l'editor di testo col codice del programma, nell'altra l'interprete Python mostra il risultato dell'esecuzione del programma.

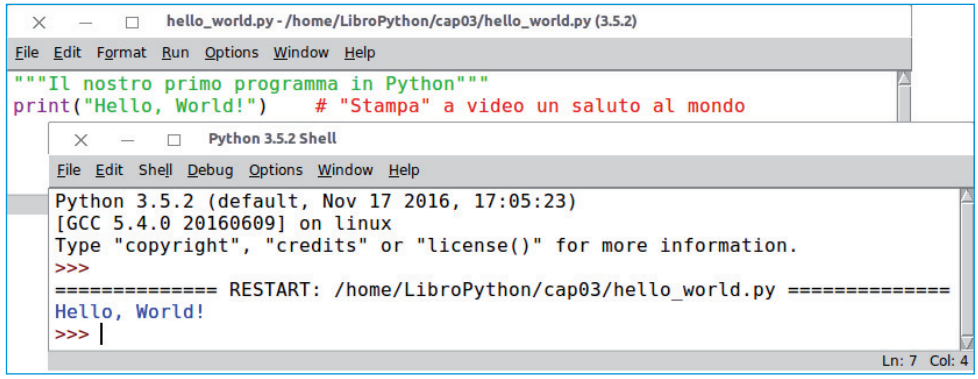


Figura 3.1 Codice e risultato dell'esecuzione del programma hello_world.py su Linux; su Windows e Mac OS X cambia solo la scritta iniziale.

Il comando **print()** (dall'inglese *to print*, stampare) visualizza la stringa di testo indicata.

Apertura in Windows dei file Python

Facendo doppio clic su un file con estensione **.py**, Windows potrebbe aprire un editor di testo come Notepad. Questo è dovuto alla mancanza di associazione tra questa estensione e il programma **python.exe**. In questi casi, fai clic col tasto destro sull'icona del file e aprilo con Python, che, a sua volta, apre la finestra con sfondo nero del terminale a carattere per eseguire il programma. Al termine dell'esecuzione sia la finestra del terminale che le eventuali finestre aperte da Python (per esempio quella di Turtle che vedremo più avanti) si chiudono automaticamente. Un modo alternativo è quello di lanciare IDLE e aprire ed eseguire da qui il programma Python. Puoi anche associare i file Python a IDLE. Per farlo, fai clic con il tasto destro del mouse sul file Python e scegli *Apri con* e poi *Scegli un'altra app*. Nella finestra che compare seleziona *Usa sempre questa app per aprire file .py*, fai clic su *Altre app* e poi, in fondo all'elenco, su *Scegli un'altra app in questo PC*.

Nella finestra che si apre seleziona nella directory di installazione di Python (nel nostro caso è `C:\Users\Maurizio\AppData\Local\Programs\Python\Python36-32\Lib\idlelib`) il file **idle.bat**.

I commenti in Python

I commenti servono agli esseri umani per spiegare e capire il codice e sono completamente ignorati dall'interprete Python. Nel programma precedente sono presenti due tipi di commento. Il primo utilizza una stringa multilinea (in questo caso la stringa è su una sola linea) per commentare il programma e il file (o modulo) che lo contiene. Il secondo tipo di commento è introdotto dal simbolo # (cancelletto, o *pound*), termina alla fine della riga e descrive una singola istruzione o una breve sequenza di istruzioni. Nei programmi è buona norma commentare solo ciò che non è già chiaro o ovvio.

Terminazione di un programma Python

Un programma Python termina dopo aver eseguito l'ultima istruzione oppure quando incontra il comando **exit()**. Se il programma ha una finestra associata puoi forzarne la chiusura chiudendo quest'ultima. Se il programma è stato lanciato nell'interprete puoi bloccare la sua esecuzione con la combinazione di tasti CRTL+C.

Perché iniziare con un saluto al mondo

La tradizione di scrivere il primo programma che dà in output "Hello, world!" risale agli informatici statunitensi Brian Kernighan e Dennis Ritchie e al loro famoso libro *Il linguaggio C*. Nei confronti di Ritchie, morto nel 2011, siamo debitori, oltre che per la creazione del linguaggio C (da cui ha attinto anche Python), per il sistema operativo Unix, il progenitore di Linux, Mac OS X e Android.

Personalizzare IDLE

Puoi personalizzare IDLE scegliendo la voce *Configure IDLE* nel menu *Options*. Per lavorare in modo più agile, effettua le seguenti modifiche.

- Seleziona *No Prompt* nella scheda *General*, in questo modo quando modifichi e lanci un programma IDLE lo salvi automaticamente senza dover passare dalla finestra di conferma riportata nella Figura 3.2.

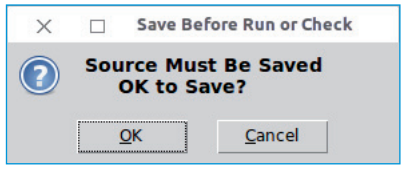


Figura 3.2 Schermata di conferma di Salva prima di eseguire.

- Imposta gli *shortcut* (scorciatoie da tastiera), *Up Arrow* (freccia su) e *Down Arrow* (freccia giù), per muoverti nell'*history* (storia dei comandi) e recuperare facilmente i comandi precedentemente impartiti nella sessione corrente dell'interprete, senza doverli riscrivere. Nella Figura 3.3 sono indicati i passaggi per creare il primo shortcut partendo dalla scheda *Keys* (tasti). Dopo aver dato l'*OK*, salva con un nome a piacere il *New Custom Key Set* (nuovo insieme di tasti personalizzato).

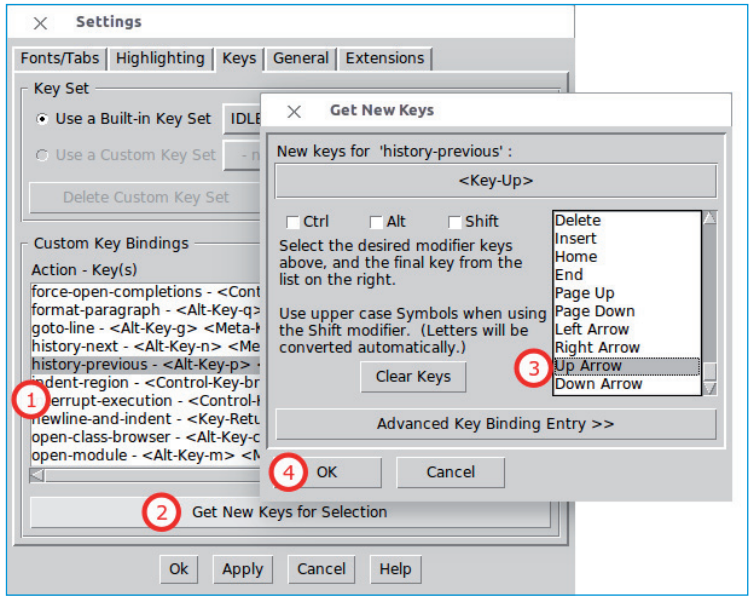


Figura 3.3 Impostazione del tasto Up Arrow per attivare la funzione history-previous e recuperare i comandi precedentemente eseguiti, premendo nell'interprete la freccia verso l'alto.

NOTA

Su `http://idlex.sourceforge.net/` trovi una ventina di estensioni per IDLE, che migliorano alcune funzionalità del programma e ne aggiungono di nuove. Tra queste ce n'è una che aggiunge, nella finestra dei programmi, una barra di numerazione delle linee utile, per esempio, quando Python segnala in una certa linea la presenza di un errore nel codice (nella barra di stato in basso a destra, IDLE riporta comunque numero di linea e colonna dove si trova il cursore di inserimento).

Valutazione di un'espressione

Quando si dialoga con l'interprete, inserendo un dato o un'espressione in forma appropriata e premendo il tasto Invio, viene riportato il risultato della sua valutazione. Nei programmi veri e propri, per avere un output a video è necessario utilizzare sempre **print()**.

Vediamo la differenza tra il chiedere all'interprete di valutare una stringa e stampare a video il suo valore:

```
>>> "ciao"
"ciao"
>>> print("ciao")
ciao
```

Nel primo caso, l'output riporta anche i delimitatori (il fatto che vengano usati gli apici o le virgolette è indifferente) ed è quindi leggermente diverso da quanto si ottiene col comando **print**.Vediamo un esempio di valutazione di stringa multilinea (il carattere **\n** rappresenta il carattere di controllo "a capo"):

```
>>> """Questa è una stringa
su più linee"""
"Questa è una stringa\nsu più linee"
```

Per i numeri non si usano delimitatori:

```
>>> 113
113
```

LSEP: Leggi, Scrivi, Esegui e Pensa

Non limitarti a leggere il codice. È essenziale anche scriverlo, eseguirlo, vedere il risultato e chiedersi perché Python "ragioni" in un certo modo e non in un altro. In fondo si tratta di applicare il *metodo scientifico* alla programmazione: effettuare ipotesi, sperimentare, valutare i risultati, introdurre piccole variazioni nel codice o nei dati e ricominciare daccapo. Python è uno strumento ideale per applicare questo metodo grazie, in particolare, alla presenza dell'interprete con cui sperimentare "al volo" parti di codice.

Operazioni aritmetiche ed espressioni

Il sistema di elaborazione elettronico all'inizio era chiamato "calcolatore" perché veniva utilizzato inizialmente per calcoli di vario tipo, come quelli relativi alle traiettorie balistiche e missilistiche.

Vediamo come utilizzare il PC, grazie all'interfaccia offerta da Python come una potente calcolatrice con la possibilità di utilizzare valori, variabili, operatori e funzioni predefinite per costruire espressioni semplici e complesse.

Addizione, sottrazione, moltiplicazione e divisione

Partiamo dalle quattro operazioni aritmetiche di base: addizione, sottrazione, moltiplicazione e divisione, che operano su due numeri, gli operandi, e producono un terzo numero, il risultato.

In IDLE prova il codice riportato di seguito (separa con uno spazio gli operandi dagli operatori per rendere il codice più leggibile). Come abbiamo già detto, non limitarti a leggere, per imparare a programmare è d'obbligo fare pratica e anche sbagliare, infatti, "sbagliando s'impara", purché ci si ragioni sopra.

```
>>> 3 + 5       # Addizione
8
>>> 7 - 4       # Sottrazione
3
>>> 9 * 2       # Moltiplicazione
18
>>> 9 / 2       # Divisione
4.5
```

L'operatore di moltiplicazione usa il simbolo asterisco (*), mentre la barra (/) rappresenta l'operatore di divisione.

Dividendo 9 per 2 otteniamo il numero decimale 4,5 ma, dal momento che la programmazione è nata nel mondo anglosassone, è prevalso l'uso della notazione americana che utilizza il "punto" come separatore decimale tra la parte intera e la parte frazionaria, per cui abbiamo 4.5.

Espressioni aritmetiche e ordine di precedenza

Come nella matematica che si impara a scuola, anche in Python puoi costruire espressioni aritmetiche combinando tra loro più operazioni. Un'espressione può essere composta da valori, operatori, parentesi, variabili e funzioni (che vedrai tra breve). Nei commenti abbiamo riportato i passaggi che portano al valore finale.

```
>>> 2 + 4 - 9    # → 6 - 9 → -3
-3
>>> 4 - 9 + 2    # → -5 + 2 → -3
-3
```

Con l'addizione e la sottrazione non conta l'ordine in cui vengono eseguite le operazioni perché, per la proprietà associativa, il risultato non cambia. Se, oltre a queste operazioni, sono presenti nell'espressione anche moltiplicazioni e divisioni le cose cambiano, perché queste ultime hanno la precedenza sulle prime.

In generale, per la valutazione di espressioni composte da più operazioni l'ordine è il seguente:

1. si parte risolvendo le espressioni contenute nelle coppie di parentesi più interne e mano a mano si procede verso l'esterno;
2. si segue l'ordine di precedenza degli operatori di Python, che è sostanzialmente quello che si impara a scuola (la moltiplicazione prima della somma, ecc.);
3. infine, a parità di ordine di precedenza, si valutano le operazioni da sinistra a destra.

Nel prossimo esempio Python esegue prima l'operazione di moltiplicazione e poi somma il risultato 12 al valore 2 per arrivare al valore finale 14:

```
>>> 2 + 4 * 3     # → 2 + 12 → 14
14
```

Le parentesi cambiano l'ordine di precedenza, assicurando che un'operazione sia eseguita prima di un'altra (ricordati sempre di bilanciare le parentesi: a ogni parentesi aperta deve corrispondere la relativa parentesi chiusa):

```
>>> (2 + 4) * 3     # → 6 * 3 → 18
18
```

Nella programmazione, diversamente dalla matematica, si usano sempre le parentesi tonde anche se ci sono più livelli di parentesi:

```
>>>  24 / ((2 + 4) * 3)     # → 24 / (6 * 3) → 24 / 18 → 1.3333333333333333
1.3333333333333333
```

Potenza, divisione intera e modulo

In matematica, l'operazione di potenza moltiplica un numero, chiamato base, per se stesso tante volte quanto è specificato da un altro numero, chiamato esponente. In generale, con base "b" ed esponente "n" abbiamo:

$$b^n = \underbrace{b \cdot b \cdots b}_{n \ volte}$$

Per esempio, 2 (la base) elevato alla quinta (l'esponente) è uguale a 32, poiché:

$2^5 = 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 32$

L'elevamento alla seconda e alla terza si chiamano anche, rispettivamente, quadrato e cubo di un numero; quindi il cubo di 4 è 64, infatti: $4^3 = 4 \cdot 4 \cdot 4 = 64$.

In Python l'operatore di elevamento a potenza è rappresentato dal doppio asterisco **:

```
>>> 4 ** 3    # Elevamento a potenza: → 4 * 4 * 4 → 16 * 4 → 64
64
```

Per la *divisione intera*, che considera solo il quoziente come risultato, l'operatore è la doppia barra **//**, mentre per il resto della divisione intera si usa l'operatore *modulo* rappresentato dal segno di percentuale **%**.
Per esempio, 7 diviso 3 fa 2 col resto di 1 (si dice anche: il 3 nel 7 ci sta 2 volte col resto di 1).

```
>>> 7 / 3     # Divisione "normale"
2.3333333333333335
>>> 7 // 3    # Divisione intera
2
>>> 7 % 3     # Resto della divisione intera
1
```

Variabili e operatore di assegnamento

Una variabile identifica una locazione (o cella) nella memoria di lavoro del computer (o memoria RAM), che memorizza un dato e alla quale ci si può riferire attraverso un nome.

Possiamo considerare una variabile come un contenitore, una scatola o un cassetto, con scritto un nome sull'esterno e con un contenuto, un valore, all'interno (Figura 3.4). Si chiama "variabile" perché il valore contenuto può variare nel tempo, cioè nel corso del programma.

Il nome di una variabile identifica univocamente un contenitore specifico.

Per memorizzare un valore in una variabile si utilizza l'operatore di assegnazione rappresentato dal simbolo di uguale (=). Vediamo, per esempio, il codice per creare una variabile chiamata **a** e assegnarle il valore 5 e una variabile di nome **b** e assegnarle il valore 8:

```
>>> a = 5    # Creazione di "a" e assegnazione del valore 5
>>> b = 8    # Creazione di "b" e assegnazione del valore 8
```

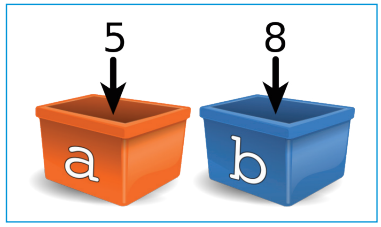


Figura 3.4 Assegnazione alle variabili a e b rispettivamente dei valori 5 e 8.

Dopo questa istruzione, i nomi **a** e **b** vengono aggiunti al vocabolario (cioè, all'insieme delle parole valide, o conosciute, del linguaggio) di Python per la corrente sessione di lavoro. Si tratta, quindi, di un vocabolario dinamico, che, oltre alle parole predefinite del linguaggio come per esempio "print", include mano a mano le parole che definiamo nell'interprete e nel programma.

Alla chiusura del programma le istruzioni impartite all'interprete e le variabili con il loro contenuto vengono perse, dimenticate (questo è la ragione per cui devi salvare su file il codice che vuoi eseguire anche in futuro).

Quando Python analizza un'istruzione e incontra una variabile la sostituisce col suo valore:

```
>>> a
5
>>> b
8
>>> a * 2
10
>>> a + b
13
```

Assegnando un nuovo valore a una variabile, sostituiamo, e perdiamo, il valore che vi era contenuto precedentemente:

```
>>> a = 2       # Modifica il valore di "a", assegnandole il nuovo valore 2
>>> a
2
```

L'operatore di assegnamento, o assegnazione, utilizza lo stesso simbolo, l'uguale, usato nelle equazioni in matematica, ma opera in modo differente. Infatti, lo scambio degli operandi genera un errore di sintassi:

```
>>> 2 = a
SyntaxError: can't assign to literal
```

Python segnala che non puoi assegnare un valore al 2, poiché quest'ultimo non è un contenitore ma un *letterale*, cioè un valore costante (altrimenti potremmo scrivere un'istruzione nella quale diciamo per esempio: "D'ora in poi fai in modo che il 3 valga 5", ma il 3 vale e deve sempre valere "letteralmente" 3). Si ha lo stesso errore mettendo a sinistra un'espressione, visto che questa viene valutata e risolta in un valore. Quindi, anche l'istruzione seguente non ha senso:

```
>>> 2 * a = 4
SyntaxError: can't assign to literal
```

Operatori di assegnamento composto

Spesso capita di dover incrementare il valore di una variabile. Un primo modo per farlo è quello di utilizzare gli operatori di somma e assegnamento.

Per esempio, per incrementare di 1 il valore della variabile "a" basta scrivere:

```
>>> a = a + 1     # Aggiunge 1 ad "a"
```

Un altro modo equivalente è quello di ricorrere all'operatore di incremento, rappresentato dall'operatore di assegnamento composto **+=**, che significa "aggiungi" alla variabile di sinistra il valore dell'operando di destra.

```
>>> a += 1     # Aggiunge 1 ad "a"
```

Python mette a disposizione operatori di assegnamento composto per tutte le operazioni viste in precedenza. Per esempio, per diminuire di 2 il valore della variabile **a** puoi scrivere:

```
>>> a -= 2     # Decrementa il valore di "a" di 2
```

Le funzioni predefinite

Abbiamo già visto la funzione predefinita **print()**, che stampa a video il contenuto inserito all'interno delle parentesi. Vediamo ora alcune funzioni matematiche predefinite che, come in matematica, ricevono un valore in input (tra parentesi) e restituiscono un valore in output.

NOTA

Non confondere l'output a video con l'output di una funzione. Nel primo caso, l'output è rivolto all'utente e visualizza un testo sullo schermo. Nel secondo caso, l'output è il risultato ritornato da una funzione ed è un valore che può essere memorizzato in una variabile, utilizzato in un'espressione e, eventualmente, anche visualizzato a video con la funzione **print()**.

Le funzioni predefinite (visualizzate in violetto da IDLE) **max()**, **min()** e **abs()** ritornano rispettivamente il massimo e il minimo in una lista di valori separati da virgole e il valore assoluto *(absolute value)* di un numero, cioè il suo valore senza segno (per esempio, ritorna 3 sia con 3 che con -3):

```
>>> max(3, 2, 7, 8, 4)
8
```

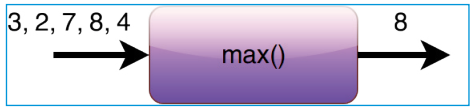


Figura 3.5 La funzione max(), dato un elenco di valori, ritorna il valore più grande.

```
>>> min(3, 2, 7, 8, 4)
2
>>> abs(3)
3
```

```
>>> abs(-3)
3
```

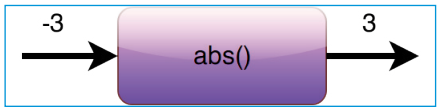
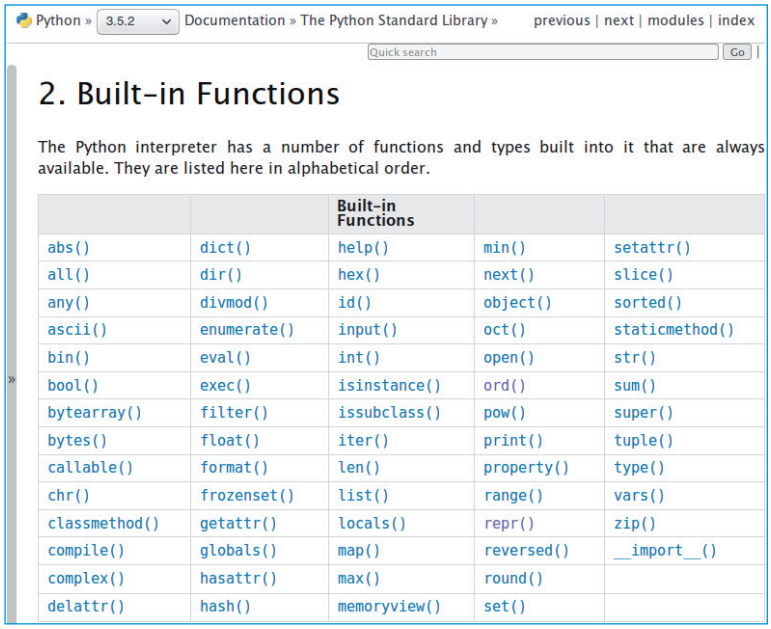


Figura 3.6 La funzione abs(), dato un valore numerico, ritorna il corrispondente valore senza segno.

Le funzioni predefinite (incorporate o *built-in*) di Python sono più di 60. Per un elenco completo e una loro descrizione in inglese, puoi consultare la documentazione ufficiale all'indirizzo `https://docs.python.org/3/library/functions.html` (Figura 3.7).



Python » 3.5.2 Documentation » The Python Standard Library » previous | next | modules | index

Quick search Go

2. Built-in Functions

The Python interpreter has a number of functions and types built into it that are always available. They are listed here in alphabetical order.

		Built-in Functions		
abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

Figura 3.7 Elenco delle funzioni predefinite di Python nella pagina della documentazione ufficiale.

Call tip: suggerimenti per le funzioni

In IDLE, sia nell'interprete Python che nella finestra dell'editor, dopo aver digitato il nome di una funzione e la parentesi aperta **(**, appare un riquadro con un suggerimento (chiamato *call tip*) che aiuta a capire cosa fa la funzione e come può essere usata.

Come puoi vedere nella Figura 3.8, scrivendo **abs(** il call tip suggerisce che tale funzione restituisce il valore assoluto dell'argomento, cioè del valore fornito in input alla funzione e inserito all'interno delle parentesi.

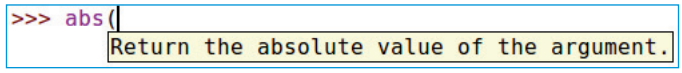


Figura 3.8 Mentre scrivi una funzione, IDLE dà un suggerimento su come usarla.

I tipi di dato

Un famoso libro scritto nel 1976 dall'informatico svizzero Niklahus Wirt (famoso soprattutto per aver creato il linguaggio di programmazione Pascal) si intitola "Algoritmi + Strutture Dati = Programmi". L'espressione rende bene il fatto che l'informatica si occupa di governare e trasformare dati e che il lavoro del programmatore è quello di trovare:

- le modalità per rappresentare e strutturare i dati, definendo i contenitori che servono per i valori dei dati di ingresso, dei valori intermedi e dei risultati finali;
- i procedimenti, o gli algoritmi, che, operando sui dati iniziali, permettono di arrivare al risultato finale.

Un tipo di dato, o classe di valori, è definito da:

- un nome;
- un insieme di valori ammissibili;
- un insieme di operazioni ammissibili su tali valori;
- una quantità, o dimensione, di memoria (in byte) per contenere un singolo valore;
- un sistema di rappresentazione, o codifica, per i valori ammissibili;
- una sintassi di lettura e una di scrittura dei valori.

In particolare, considerando gli interi, ci interessa sapere quali valori interi possiamo rappresentare e quali operazioni possiamo effettuare sugli interi.

In Python la funzione predefinita **type()** ritorna il tipo di dato di un valore o di una variabile, cioè del valore contenuto in quest'ultima.

Vediamo ora più dettagliatamente i due tipi di dato che Python mette a disposizione per rappresentare i numeri.

I tipi di dato numerici int e float

Per rappresentare valori numerici Python prevede i tipi di dato **int** e **float**. Il primo permette la rappresentazione di numeri interi, mentre il secondo rappresenta valori con la virgola (*floating point* vuol dire "a virgola mobile" e indica un formato di rappresentazione di numeri con parte intera e parte frazionaria), ma memorizza solo un numero limitato di cifre significative per la parte frazionaria.

```
>>> type(218) # Ritorna il tipo (la classe "int") associato al valore 218
<class "int">
>>> n = 76
>>> type(n) # Tipo associato al valore contenuto in n
<class "int">
>>> type(3.82) # Tipo (la classe "float") del valore 3.82
<class "float">
>>> x = 844.2913
>>> type(x) # Tipo (del valore) della variabile x
<class "float">
>>> 1 / 3 # In questo caso abbiamo 16 cifre dopo la virgola
0.3333333333333333
>>> 3 / 7 # In questo caso abbiamo 17 cifre dopo la virgola
0.42857142857142855
```

Come abbiamo visto:

```
>>> 6 / 2 # Il risultato della divisione è un numero con la virgola
3.0
>>> 6 // 2 # Il risultato della divisione intera è un intero
3
```

Una particolarità di Python è quella di non avere un limite (se non quello legato alla memoria del PC in uso) sulla dimensione per i numeri interi. Per verificarlo prova la seguente operazione:

```
>>> 1234567890 ** 10000 # Un elevamento a potenza piuttosto grande!
```

La funzione print()

Approfondiamo alcuni aspetti della funzione **print()**, analizzando una parte del suo call tip (Figura 3.9).

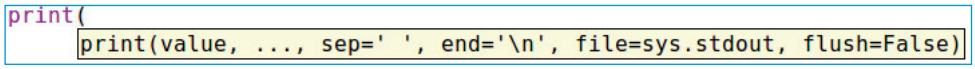


Figura 3.9 Call tip della funzione print().

Passando in ingresso una lista di valori (*value*) separati da virgole, **print()** li stampa separandoli con uno spazio (in informatica, lo spazio si chiama anche *blank*) e alla fine (*end*) va a capo (**\n** indica il carattere di controllo *new line*).

Quindi, la scrittura **end='\n'** indica che, di default (cioè, in mancanza di diversa specificazione), il comportamento predefinito della funzione **print()** è quello di terminare con la stampa del carattere speciale **\n**, che porta il cursore sulla riga successiva:

```
>>> print(3, 12.5, 8, 1.0, 7)
3 12.5 8 1.0 7
```

Puoi modificare il comportamento di default, o predefinito, della funzione impostando i valori per il separatore (**sep**) e per il terminatore (**end**):

```
>>> print(3, 1.0, 7, sep=", ")    # Usa come separatore una virgola e uno spazio
3, 1.0, 7
>>> print(2, 3, end=" fine riga") # Alla fine stampa la stringa " fine riga"
2 3 fine riga
```

Crea ora il nuovo programma riportato nel listato seguente.

stampa_operazione_e_risultato.py

```
"""Due modi equivalenti per stampare il testo di un"operazione e il suo
risultato"""
# 1° modo: usa un'unica istruzione per stampare l'operazione e il risultato
print("2^8 =", 2 ** 8)

# 2° modo: usa due istruzioni per stampare l'operazione e il risultato
print("2^8 =", end=" ") # Dopo la stampa non va a capo, ma stampa uno spazio
print(2 ** 8)
```

Secondo una convenzione matematica diffusa, abbiamo usato il carattere ^ per indicare, nella stringa di descrizione, l'operazione di elevamento a potenza, mentre per l'esecuzione abbiamo utilizzato la sintassi di Python:

```
2^8 = 256
2^8 = 256
```

Per qualche calcolo in più…

Puoi aggiungere funzionalità al computer programmandolo. Per iniziare a farlo, affrontiamo un paio di problemi di matematica finanziaria e di fisica.

Calcolo dell'interesse composto

Mettiamo alla prova la *natura di calcolatrice estesa* del computer risolvendo un classico problema di economia: "Se depositiamo in banca 100 € a un tasso di interesse annuo del 3% quanti soldi avremo dopo 10 anni?"

All'inizio abbiamo 100 €, dopo un anno si aggiungono i 3 € di interesse portando il nostro capitale a 103 €, dopo 2 anni maturano altri 3,09 € di interesse (infatti, 103 · 0,03 = 3,09) e il capitale diventa 106,09 € e così via.

I dati di ingresso con cui abbiamo a che fare sono i seguenti:

- *capitale iniziale* (C_i);
- *capitale finale* (C_f), in economia si chiama anche "montante", perché nel corso degli anni, se il tasso di interesse è positivo, il suo valore aumenta;

- *tasso di interesse annuo* (*tasso*), è la percentuale di interesse che la banca riconosce dopo un anno di deposito del capitale;
- *numero di anni* (n) di deposito.

Vedremo nei prossimi capitoli un programma che risolve il problema e stampa la crescita del capitale anno per anno. Per ora utilizziamo la seguente *formula per il calcolo dell'interesse composto* (si chiama *composto* perché l'interesse si calcola sia sul capitale iniziale che sull'interesse accumulato fino all'anno precedente):

$C_f = C_i(1 + tasso)^n$

Input, elaborazione, output

Crea il nuovo programma **interesse_composto.py**. Nel codice sono separate e commentate le tre funzionalità che compongono il programma:

- input e impostazione dei dati iniziali;
- elaborazione;
- output dei dati finali.

interesse_composto.py

```
"""Calcolo dell"interesse composto"""
# Dati iniziali
capitale_iniziale = 100
tasso = 0.03    # Corrisponde al 3%
n_anni = 10

# Elaborazione
capitale_finale = capitale_iniziale * (1 + tasso) ** n_anni

# Output
print("Dopo", n_anni, "anni il capitale è:", capitale_finale)
print("Dopo", n_anni, "anni il capitale è:", round(capitale_finale, 2))
```

L'esecuzione fornisce il seguente output a video:

```
Dopo 10 anni il capitale è: 134.39163793441222
Dopo 10 anni il capitale è: 134.39
```

In molte occasioni la separazione "input/elaborazione/output" non è possibile, perché nella sequenza logica e d'uso del programma da parte dell'utente, si intervallano e si intersecano istruzioni relative a tutte e tre queste funzionalità.

Nell'ultima istruzione, abbiamo passato alla funzione **print()** il valore di "capitale_finale" arrotondato alla seconda cifra decimale grazie alla funzione predefinita **round()**. Non indicando il secondo argomento, **round()** arrotonda il valore all'intero più vicino.

Vediamo con qualche esempio, la differenza tra l'uso di **round()** con zero cifre decimali e l'uso della funzione predefinita **int()**, che trasforma un valore in un numero intero troncando, cioè rimuovendo, la sua parte frazionaria:

```
>>> int(17.45)
17
>>> round(17.45)    # Arrotonda il valore all'intero più vicino 17
17
>>> int(17.84)
17
>>> round(17.84)    # Arrotonda il valore all'intero più vicino 18
18
```

Calcolare la velocità

"Oggi ho fatto 4,1 km di corsa in 21 minuti 34 secondi. Qual è stata la mia velocità oraria e in metri al secondo?"

Per saperlo crea il file **velocita_in_kmh_e_ms.py** col codice riportato nel prossimo listato.

Nomi di file

Per i nomi di file è buona norma iniziare con una lettera, e poi utilizzare solo lettere e cifre e il carattere di sottolineatura (*underscore*) per separare tra loro le parole. Inoltre, evita le lettere accentate, perché non sono presenti nella lingua inglese, che rimane un riferimento importante per i linguaggi di programmazione.

Per calcolare la velocità in metri al secondo (m/s) trasformiamo il tempo impiegato in secondi, salvandolo nella variabile intermedia **secondi_totali**, mentre per avere i chilometri all'ora (km/h) sfruttiamo questa proporzione:

<km percorsi> : <km all'ora> = <secondi impiegati> : <secondi in un'ora>

velocita_in_kmh_e_ms.py

```
"""Calcolo della velocità in km/h e m/s"""
# Dati iniziali
minuti = 21     # Tempo impiegato: 21" e 34""
secondi = 34
km = 4.1        # Distanza percorsa: 4,1 km

# Elaborazione
secondi_totali = minuti * 60 + secondi

# Velocità in m/s
metri = km * 1000
m_s = metri / secondi_totali

# Velocità in km/h. Uso la seguente proporzione (3600 sono i secondi in un ora):
# km : km_h = secondi_totali : 3600
km_h = km * 3600 / secondi_totali
```

```
# Output
print("Velocità:", round(m_s, 2), "m/s")
print("Velocità:", round(km_h, 2), "km/h")
```

Il risultato è questo:

```
Velocità: 3.17 m/s
Velocità: 11.41 km/h
```

Stile di programmazione

È essenziale fare in modo che i nostri programmi possano essere letti e capiti facilmente. Infatti, come recita la massima della Figura 3.10 tratta dallo "Zen di Python", la leggibilità è importante.

Figura 3.10 "La leggibilità è importante". "Davvero?", illustrazione di Karlisson M. Bezerra (http://hacktoon.com/log/2016/the-zen-of-python-illustrated/).

Nomi validi e autodocumentanti per gli identificatori

La scelta di quali nomi utilizzare per identificare gli oggetti del programma (variabili, funzioni, ecc.) spetta al programmatore. Python controlla che siano rispettate alcune regole e non si interessa del loro significato (nei programmi precedenti avremmo potuto tranquillamente scegliere come nomi "x", "y" e "z" oppure "pippo", "pluto" e "paperino").

Un altro dei motti di Python è quello raffigurato nella Figura 3.11: "Esplicito è meglio di implicito". Un identificatore dovrebbe avere un nome che ne riveli l'identità, lo scopo e il significato in modo chiaro ed esplicito, così che chi legge il codice non debba perdere troppo tempo chiedendosi "cos'è?" e "a che cosa serve?".

Per gli identificatori non è opportuno scegliere nomi di parole riservate, cioè già utilizzate da Python, come per esempio "if" oppure "and", e di funzioni predefi-

nite come "print" e "max". Infine, lo ricordiamo ancora, è utile scegliere "nomi autodocumentanti", cioè che non siano troppo lunghi e che esprimano con chiarezza il significato di ciò che identificano all'interno del programma.

Figura 3.11 Un difficile riconoscimento: Chi sei?, Non ho nome, non ho scopo sono la variabile x.

Nomi validi per gli identificatori

Un identificatore, o nome di un oggetto, in Python si compone di lettere, cifre o l'underscore, ma deve iniziare con una lettera o un underscore. Utilizza l'underscore anche per separare tra loro le parole all'interno di un identificatore come in "capitale_iniziale".

Il documento PEP8

Guido van Rossum ha pubblicato il documento *PEP8* per definire alcune regole di stile per la stesura di codice Python (la traduzione in italiano curata da Alex Martelli è disponibile su `http://www.python.it/doc/articoli/pep-8.html`).

Nel libro, a parte alcune "licenze didattiche", abbiamo cercato di attenerci a tali indicazioni.

Due regole di stile, che affronteremo meglio più avanti, sono:

- separare gli operatori dagli operandi con uno spazio;
- inserire una riga vuota (non più di una), dopo le istruzioni iniziali di import, prima di un commento, una selezione o un ciclo, a meno che non vi sia un'*indentazione* che rende già chiaro lo stacco.

Help!

Quando non ricordi qual è la sintassi di un'istruzione, di una funzione, di una libreria, di un oggetto o di un valore, puoi ricorrere alla funzione predefinita **help()**, che stampa una descrizione più esaustiva di quella data dal call tip di IDLE.

Vediamo un esempio d'uso di **help()** nell'interprete con la funzione **round()** appena vista:

```
>>> help(round)
Help on built-in function round in module builtins:

round(...)
    round(number[, ndigits]) -> number

    Round a number to a given precision in decimal digits (default 0 digits).
    This returns an int when called with one argument, otherwise the
    same type as the number. ndigits may be negative.
```

L'help è scritto in inglese (buona occasione per impararlo o migliorarlo) e riporta tre informazioni su "round":

- è una funzione predefinita che appartiene alla libreria (o modulo, *module*) *built-in* (Figura 3.7);
- la sintassi per richiamarla prevede un primo argomento numerico (*number*) obbligatorio e un secondo argomento opzionale (*ndigits*, numero di cifre); nella convenzione utilizzata per descrivere la sintassi dei comandi si utilizzano le parentesi quadre per indicare l'opzionalità;
- restituisce un numero arrotondato, specificando il numero di cifre dopo la virgola. Se non viene indicato un valore per **ndigits** il valore di default è zero, cioè senza alcuna cifra decimale.

Help online e offline

Un aiuto online, come quello fornito dalla funzione **help()**, è disponibile solo attraverso un dispositivo e un'applicazione informatica. Al contrario, un aiuto offline è disponibile anche a computer spento: per esempio è quello che può dare un libro oppure una lavagna in ardesia (non la LIM, o *Lavagna Interattiva Multimediale*).

Proposte di variazione sul tema

1. Scrivi il programma **quadrato_e_cubo.py** che definisce la variabile **n**, assegnandole il valore 7, e stampa il suo quadrato e il suo cubo. L'esecuzione fornisce il seguente output:

   ```
   Il quadrato di 7 è: 49
   Il cubo di 7 è: 343
   ```

2. Scrivi il programma **interesse_semplice.py** che, dopo avere definito le variabili con gli stessi dati iniziali del programma **interesse_composto.py**, calcola e stampa l'interesse semplice, che non considera l'interesse sull'interesse ma solo sul capitale iniziale ed è definito dalla formula:

   capitale_finale = capitale_iniziale · (1 + tasso · n_anni)

L'esecuzione fornisce il seguente output (in questo caso, essendo nulla la parte frazionaria, la funzione **round()** ritorna un solo zero anche specificando due o più cifre decimali):

```
Dopo 10 anni il capitale è: 130.0
```

3. Il miglio terrestre è un'unità di misura di lunghezza del sistema britannico che corrisponde a 1.609,344 metri. Scrivi il programma **velocita_in_mph.py** che, dato un tempo di percorrenza e una distanza, calcola la corrispondente velocità in m/s, in km/h e in mph (*miles per hour* o miglia all'ora). Il risultato, con i dati del programma visto poco sopra **velocita_in_km-h_e_m-s.py**, è il seguente:

```
Velocità: 3.17 m/s
Velocità: 11.41 km/h
Velocità: 7.09 mph
```

4. Crea il programma **conversione_temperature.py** per la conversione di valori di temperatura tra gradi Celsius (o centigradi) e gradi Fahrenheit e Kelvin. Le formule di equivalenza sono:

   $T(°F) = T(°C) \cdot 1.8 + 32$

   $T(°K) = T(°C) + 273.15$

   Eseguendo il programma e inserendo il valore 30 otteniamo:

```
Temperatura in °C: 30
Temperatura in °F: 86.0
Temperatura in °K: 303.15
```

Cosa hai imparato

- A scrivere il tuo primo programma.
- Come creare stringhe normali e su più linee.
- Come usare i commenti su una linea e su più linee.
- Gli operatori base dell'aritmetica e altri operatori secondari quali il modulo e l'elevamento a potenza.
- La scrittura di espressioni aritmetiche e le regole di precedenza nella loro valutazione.
- Come personalizzare IDLE.
- Che cosa sono e come si usano le variabili.
- L'operatore di assegnamento semplice e composto.
- L'uso di alcune funzioni matematiche predefinite.
- Un uso più approfondito della funzione **print()**.
- Come calcolare l'interesse composto.
- Come calcolare la velocità in varie unità di misura.