Capitolo 1

Cos'è Python?

La domanda è, non a caso, la prima delle FAQ (Frequently Asked Questions) ovvero le domande più frequenti, presenti sul sito ufficiale di Python, `http://www.python.org`. Leggiamo insieme la risposta e non preoccupiamoci se qualche termine ci sembrerà un po' criptico, perché nei prossimi paragrafi e capitoli avremo modo di affrontare in dettaglio ogni definizione:

"*Python è un linguaggio di programmazione interpretato, interattivo e orientato agli oggetti. Incorpora al proprio interno moduli, eccezioni, tipizzazione dinamica, tipi di dati di altissimo livello e classi. Python combina un'eccezionale potenza con una sintassi estremamente chiara. Ha interfacce verso molte chiamate di sistema, oltre che verso diversi ambienti grafici, ed è estensibile in C e in C++. Inoltre è usabile come linguaggio di configurazione e di estensione per le applicazioni che richiedono un'interfaccia programmabile. Da ultimo, Python è portabile: può girare su molte varianti di Unix, su Mac, su PC con MS-DOS, Windows, Windows NT e OS/2.*"

A chi dobbiamo una meraviglia del genere? A un geniale signore olandese che risponde al nome di Guido Van Rossum. Curiosamente Guido non è l'adattamento italiano del suo nome; è proprio il suo nome originale in olandese.

Guido, nel lontano Natale del 1989, invece di passare le vacanze a decorare l'albero, decise di scrivere un linguaggio che correggesse in gran parte, se non tutti, i difetti che secondo lui erano presenti negli altri linguaggi.

Per nostra fortuna, Guido Van Rossum era, ed è tuttora, un grandissimo esperto di linguaggi di programmazione e questo ha fatto sì che, fin da subito, la sua creatura avesse un notevole successo, dapprima tra i colleghi del centro di ricerca dove lavorava in quel periodo e poi, dopo la pubblicazione su Usenet nel febbraio del 1991, in tutto il mondo.

Qualcuno può domandarsi perché una persona decida di donare all'umanità quello che ha creato, dedicandovi così tanto del proprio

tempo libero senza ricevere in cambio altro che gratitudine. È la stessa domanda, senza risposta, che potremmo rivolgere a Linus Torvalds, autore della prima versione del kernel di Linux o, ancora meglio, ad Albert Sabin, se fosse ancora vivo, scopritore del vaccino della poliomielite, che si rifiutò sempre di brevettare.

Nel mondo libero di Internet questo accade spesso e, talvolta, si instaura un circolo virtuoso in cui le persone restituiscono qualcosa in cambio, migliorando, correggendo e diffondendo ciò che viene reso disponibile gratuitamente. Python è sicuramente un esempio lampante di questo fenomeno: attualmente esistono circa ottanta (!) sviluppatori ufficiali del linguaggio, anche se Guido Van Rossum rimane il solo e unico BDFL (*Benevolent Dictator For Life*: benevolo dittatore a vita) di Python; in altre parole è colui che ha l'ultima e definitiva parola in caso di dispute informatiche.

Il termine Python deriva dalla passione del suo ideatore per il noto gruppo di comici inglesi degli anni sessanta, i Monty Python, i quali a loro volta scelsero il proprio nome perché "suonava divertente". In un'altra FAQ si dice espressamente che nella documentazione è possibile, anzi consigliabile, far riferimento a scenette o a giochi di parole dei Monty Python.

Non è però l'unica volta che questi comici hanno dato il nome a qualcosa che avesse a che fare con l'informatica: il termine *spam*, ormai tristemente noto a chiunque abbia a che fare con la posta elettronica, deriva da un loro famosissimo sketch, in cui l'improbabile menu di un ancora più improbabile ristorante era composto da un'infinita lista di piatti, tutti invariabilmente accompagnati da montagne di spam, un tipo di carne macinata in scatola, non molto appetibile.

Vediamo ora in dettaglio le diverse definizioni contenute nella risposta alla domanda che dà il titolo a questo capitolo.

Interpretato, interattivo

Per la maggior parte dei linguaggi di programmazione, le operazioni necessarie per l'esecuzione di un programma comprendono la scrittura del codice sorgente, la compilazione, talvolta il *linkaggio* delle librerie (se non conoscete il significato di questo termine meglio per voi: con Python non serve saperlo) e infine l'esecuzione del programma eseguibile così ottenuto.

Python permette invece di eseguire direttamente il codice sorgente che avete scritto (per questo si dice che è *interpretato*) o, addirittura, di scrivere istruzioni direttamente dal suo prompt dei comandi, senza bisogno di creare o modificare un file sorgente (per questo è detto *interattivo*).

Certo quest'ultima modalità d'uso può sembrare bizzarra, ma vedremo che per iniziare da zero, per provare alcune istruzioni nuove o per testare piccole parti dei programmi, questa modalità è estremamente comoda e veloce.

Per quelli che non sanno aspettare, che hanno già scaricato e installato da soli Python, facciamo un piccolo salto in avanti per dare un rapido sguardo a un esempio di interattività.

Ecco Python in modalità a riga di comando; per maggior chiarezza (e solo in questo esempio) quello che abbiamo digitato appare in neretto, mentre il resto è il testo visualizzato dall'interprete):

```
C:\> python
Python 3.1.1 (r311:74483, Aug 17 2009, 16:45:59) [MSC v.1500 64
bit (AMD64)] on win32

Type "help", "copyright", "credits" or "license" for more
information.
>>> print("Ciao mondo!")
Ciao mondo!
>>> a = 4
>>> b = a * 2
>>> print(b)
8
>>>
```

I caratteri >>> rappresentano il prompt dell'interprete dei comandi di Python.

Nella Figura 1.1 è visibile Python in modalità finestra.

Orientato agli oggetti

Questa espressione è la traduzione letterale dell'abusatissimo termine inglese "object-oriented". Non è il caso di entrare troppo nei dettagli di questo paradigma di programmazione, su cui sono già stati versati

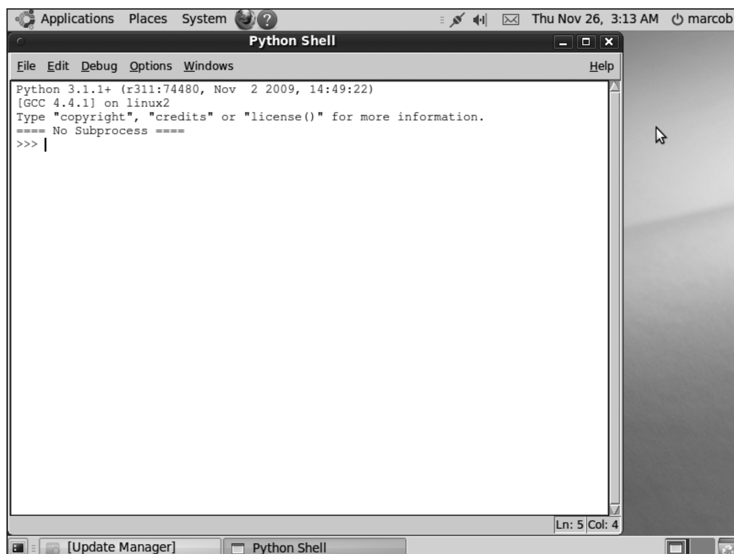

Figura 1.1 IDLE di Python nella distribuzione Ubuntu di Linux.

fiumi di inchiostro senza giungere a una definizione condivisa e universalmente accettata; possiamo dire a grandi linee che seguire questo paradigma vuol dire pensare alla soluzione di un problema non in termini di una successione di istruzioni, ma di oggetti e dei relativi attributi.

Un esempio pratico può aiutarci a comprendere meglio questo concetto. Immaginiamo di dover realizzare una libreria che renda disponibili ai nostri colleghi le tipiche funzionalità di lettura e scrittura di variabili in un file di configurazione. Usando un approccio procedurale cominceremmo a scrivere una serie di funzioni per chi dovrà usare questa libreria: `CreaFile`, `LeggiFile`, `LeggiVariabile`, `ScriviVariabile` e così via. Ogni funzione, molto probabilmente, dovrà ricevere in ingresso il nome del file, motivo per cui dovremo ricordarci il suo nome durante tutto l'uso di queste funzioni. Inoltre ogni funzione dovrà occuparsi di aprire, scrivere o leggere e chiudere il file. Ogni funzione potrà fallire per diversi motivi (file non esistente in lettura, spazio su disco terminato in scrittura e così via), per cui dovremo leggere il valore restituito e poi eventualmente andare a verificare con un'altra funzione quale sia il messaggio d'errore completo.

Con un linguaggio orientato agli oggetti possiamo affrontare il problema in maniera totalmente diversa. Per esempio possiamo definire un oggetto `FileConfigurazione`. Questo oggetto, al momento della creazione, richiederà il nome del file e da questo momento in poi non dovremo più preoccuparci di ricordarlo. Inoltre possiamo salvare all'interno dell'oggetto lo stato del file fisico, in modo da non doverlo aprire ogni volta. Il nostro oggetto fornirà le funzionalità di lettura e scrittura delle variabili, di reperimento dello stato del file e così via.

In linea teorica è possibile programmare con questo modello con molti linguaggi; ma come vedremo, con Python definire, creare e usare un oggetto è estremamente facile.

Per i curiosi ecco in Python la definizione di una classe, `Frutto`, e la creazione di una sua istanza, `mela`:

```
>>> class Frutto:
...     tipo = "vegetale"
...
>>> mela = Frutto()
>>> print(mela.tipo)
vegetale
>>>
```

La sequenza di simboli "..." indica la continuazione sulla riga seguente di un comando che richiede più istruzioni.

Moduli

Dopo aver installato Python, ci troviamo automaticamente a disposizione una grande quantità di librerie pronte per l'uso e in grado di fornire un'enorme quantità di codice, già testato e funzionante. Le librerie si chiamano *moduli*.

Come si importa un modulo in Python? Nell'esempio seguente importeremo il modulo `smtplib` e lo useremo per inviare un messaggio di posta elettronica:

```
>>> import smtplib
>>> host=smtplib.SMTP("mail.server.it")
>>> ret=host.sendmail("otello@venezia.it",
                      "desdemona@venezia.it",
                      "Cara Desdy, dov'eri ieri?")
>>>
```

Non preoccupatevi se dalla seconda istruzione in poi non vi è tutto chiaro: l'esempio vuole solo mostrare quanto è facile (anche in modalità interattiva) caricare e usare una libreria (in questo caso il modulo `smtplib`).

Eccezioni

La gestione degli errori in Python è simile a quella di altri linguaggi, per esempio Java, dato che usa il concetto di *eccezione* (`exception`). Un'eccezione è provocata da un evento anomalo o imprevisto che cambia il normale flusso d'esecuzione del codice. Un'eccezione, per esempio, può essere dovuta a un input non valido da parte di un utente (un valore alfabetico al posto di un valore numerico) oppure a un'anomalia hardware (tentare di scrivere un file su un hard disk pieno).

Le eccezioni possono essere previste dal programmatore, e in tal caso si dicono eccezioni gestite (`handled`), oppure possono essere del tutto impreviste, e in tal caso si dicono non gestite (`unhandled`).

Se non avete mai avuto modo di usare un linguaggio che preveda eccezioni, probabilmente queste poche righe non vi avranno chiarito del tutto le idee. Non allarmatevi: vista l'importanza del tema c'è un intero capitolo dedicato a questo argomento.

Per i più impazienti, ecco un esempio di eccezione non gestita in Python:

```
>>> a = 0
>>> b = 10
>>> print(b/a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: int division or modulo by zero
>>>
```

Ecco ora la stessa eccezione, questa volta gestita con l'istruzione composta `try / except`:

```
>>> a = 0
>>> b = 10
>>> try:
...     print(b/a)
... except ZeroDivisionError:
...     print("Divisione per zero")
...
```

```
Divisione per zero
>>>
```

Tipizzazione dinamica

In un linguaggio di programmazione la tipizzazione delle variabili può essere statica o dinamica. Nel primo caso il programmatore deve dichiarare esplicitamente il tipo della variabile prima di usarla. Nella *tipizzazione dinamica* è l'interprete (o il compilatore) che, in base al valore assegnato alla variabile, ne decide il tipo.

Python usa la tipizzazione dinamica, ma nonostante questo è un linguaggio fortemente tipizzato. Per esempio non è possibile sommare una variabile stringa a una variabile numerica senza convertire esplicitamente quest'ultima in una variabile stringa; se lo faceste scatenereste un'eccezione!

```
>>> a = "totale "
>>> b = 10
>>> print(a + b)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
>>>
```

Quindi si deve usare la seguente forma:

```
>>> a = "totale "
>>> b = 10
>>> print(a + str(b))
totale 10
>>>
```

Tipi di dati di alto livello

Oltre ai tipi di dati nativi più usuali che si possono usare in Python (interi, stringhe, boolean, float) ve ne sono altri estremamente specializzati: liste, tuple, set e dizionari.

Una *lista* è un elenco ordinato di oggetti non necessariamente dello stesso tipo. È possibile aggiungere alla lista nuovi elementi in fondo, all'inizio o in qualunque altra posizione, estrarre un elemento o anche una sequenza di elementi. È possibile creare cicli su tutti gli elementi, cercare un elemento e così via.

Probabilmente tutto quello che vi può venire in mente di fare con un elenco ordinato di oggetti può essere fatto facilmente con una lista.

Ecco un esempio di lista:

```
>>> a = [1, 2, 3]
>>> print(a)
[1, 2, 3]
>>> a.append("stella")
>>> print(a)
[1, 2, 3, 'stella']
>>>
```

Una *tupla* è sostanzialmente una lista immutabile: una volta assegnata non può più essere modificata. Dunque a una tupla non potete togliere o aggiungere alcun elemento. Le tuple sono utili quando i dati in esse contenuti non devono mai essere modificati, una volta che sono stati assegnati. Rispetto alle liste sono molto più efficienti per tempo di esecuzione e consumo di memoria, ma in cambio offrono molte meno funzionalità.

Ecco un esempio di tupla:

```
>>> elenco = (3, "14", 15, "92")
>>> print(len(elenco))
4
>>>
```

Un *set* è un vero e proprio insieme di elementi non ordinati e senza duplicati. Se inizializziamo un set con un elenco contenente elementi ripetuti, questi vi appariranno una sola volta. Con i set è possibile fare tutte le classiche operazioni che si possono fare con gli insiemi: unione, differenza e intersezione; esiste anche una quarta operazione, l'intersezione asimmetrica, che individua gli elementi che sono presenti in uno solo dei set.

Ecco un esempio di set:

```
>>> primi = set(["pasta", "minestra", "uova"])
>>> secondi = set(["carne", "uova"])
>>> print(primi - secondi)
{'pasta', 'minestra'}
>>> print(primi & secondi)
{'uova'}
>>>
```

Un *dizionario* è una collezione di oggetti di qualunque tipo, che possono essere reperiti tramite una chiave. La chiave può essere un intero, una stringa, persino una tupla e, in generale, qualunque oggetto immutabile (quindi non una lista). Si può anche pensare a un dizionario come a un elenco non ordinato di coppie di chiavi e valori.

Ecco un esempio di dizionario:

```
>>> anni = {'lucia': 45, 'ale': 15, 'fede': 12}
>>> print(anni['lucia'])
45
>>> print(anni.keys())
dict_keys(['fede', 'ale', 'lucia'])
>>>
```

Sintassi estremamente chiara

La sintassi di Python è talmente chiara e intuitiva che spesso si intuisce il modo giusto di usare un comando anche senza consultare la documentazione. Inoltre, leggendo un programma, anche in assenza di commenti, spesso è facile comprenderne lo scopo.

C'è una particolarità importantissima nella sintassi di Python, così importante che da sola è spesso oggetto di violente dispute tra i sostenitori e i detrattori di questo linguaggio: l'*indentazione*. Python usa l'indentazione per delimitare blocchi di istruzioni. Il linguaggio non prevede parentesi, sintassi `end-if`, `next`, `enddo` o altro: solo l'indentazione.

In questo caso un esempio è d'obbligo per tutti, non solo per i curiosi. Proviamo a leggere queste righe di codice Python:

```
if persona.anni < 18:
    print("Accesso negato")
    accesso = False
elif persona.anni > 99:
    print("Non è il caso...")
    accesso = False
else:
    accesso = True
    if persona.is_uomo():
        print("Benvenuto")
    else:
        print("Benvenuta")
return accesso
```

È evidente a colpo d'occhio dove termina il corpo di ogni istruzione `if` o di ogni istruzione `else` o `elif` (che sta per `else if`).

Questo invece è lo stesso frammento di programma scritto in C:

```
if (persona.anni < 18) {
printf("Accesso negato\n");
accesso = 0; }
else if (persona.anni > 80) {
printf("Non è il caso...\n");
accesso = 0; }
else { accesso = -1;
if (persona.isUomo == -1)
printf("Benvenuto\n");
else
printf("Benvenuta\n");
return(accesso);}
```

Siamo tutti d'accordo che in quest'ultimo caso il programmatore non è stato particolarmente ordinato, ma ha comunque scritto del codice accettabile per un compilatore C. In Python questo non si può proprio fare: non si può essere disordinati, perché il programma non funzionerebbe.

Estensibile in C e in C++

Python, pur essendo un linguaggio interpretato, è incredibilmente veloce ed efficiente. In alcuni rari casi può però servirvi tutta la potenza del processore, anche quella parte dedicata all'interprete Python, oppure volete riusare delle librerie compatibili solo con il linguaggio C o C++. Bene, in entrambi i casi è possibile, anzi facile, creare dei *moduli di estensione* per superare questi problemi. Questi moduli possono essere poi importati come ogni altro modulo standard.

Con le API (Application Programming Interface) di Python, se lo desiderate e siete particolarmente creativi, potete addirittura implementare nuovi tipi di dati pronti per l'uso nei vostri programmi.

Fate attenzione però: prima di pensare che sia necessario sviluppare un nuovo modulo di estensione, verificate su Internet che tale modulo non sia già disponibile. Tanto per fare un esempio, esiste tutta una serie di moduli per il calcolo scientifico, che mettono Python in grado di competere in termini di velocità con i programmi scritti in C o in C++. Potete reperire questi moduli e il relativo codice sorgente sul sito `http://www.scipy.org/`.

Usabile come linguaggio di configurazione

Non esiste quasi nessun programma che non preveda la possibilità da parte dell'utente di configurare in qualche modo il suo comportamento. Finché si tratta di qualche parametro, non vi sono problemi: un form dove l'utente può inserire e salvare i valori necessari è più che sufficiente.

Pensiamo invece al caso in cui il nostro applicativo, magari già scritto in un linguaggio non interpretato, debba poter essere configurato in maniera più complessa. Addirittura in alcuni casi gli utenti avanzati devono poter cambiare il comportamento del programma in base ad alcune situazioni. Il nostro programma richiede quindi quella che viene detta un'*interfaccia programmabile*. In questi casi la soluzione migliore è quella di fornire una sorta di linguaggio di *scripting*, da usare in fase di configurazione.

Purtroppo la realizzazione di un programma in grado di comprendere un linguaggio di scripting è un compito eccezionalmente oneroso. Ma Python ci viene in aiuto anche questa volta: con poche righe di codice possiamo includere il suo interprete nel nostro programma. A questo punto i file di configurazione possono contenere persino piccoli programmi scritti in Python.

Proviamo a immaginare un programma gestionale nel cui file di configurazione possiamo scrivere:

```
codici_esenti = (10, 20, 33, 44)
if codice_settore in codici_esenti:
    totale_fattura = imponibile
else:
    totale_fattura = imponibile * (1 + iva)
```

Quante release e quante installazioni di nuove versioni potremmo evitare? Ci basterà inviare il nuovo file di configurazione per email...

Portabile

È l'ultima definizione e abbiamo vita facile nel dimostrare la *portabilità* di Python. Se scriviamo un programma in Python, oltre a poterlo eseguire con il nostro sistema operativo, potremo impiegare tutti quelli più diffusi, come Windows (tutte le versioni ed esiste perfino Python per .NET: IronPython), Mac OS X e Linux (tutte le distribuzioni); possiamo perfino inviarlo perché possa essere eseguito da chi usa uno di questi sistemi operativi: AIX, AROS (Amiga Research OS), AS/400

(OS/400), BeOS, OS/2, OS/390 e z/OS, Palm OS, iPod, PlayStation e PSP (no, non è uno scherzo), Psion, QNX, RISC OS (ex Acorn), cellulari Nokia Series 60, Sparc Solaris, VMS, VxWorks, Windows CE e Pocket PC, Sharp Zaurus e MorphOS.

E se per caso il vostro sistema operativo non è presente in questo elenco, non disperate: vi basta avere un compilatore C per poter scaricare il codice sorgente e creare la vostra versione personale di Python!