Python (autore: Vittorio Albertoni)

Premessa

Nel lontano febbraio del 1991, quando ancora Internet era riservata a scienziati e ricercatori delle Università, comparve un messaggio, in inglese, i cui principali passaggi tradotti in italiano sono i seguenti:
«Ho pubblicato una versione beta del mio linguaggio Python su alt.sources.
...
Python è un linguaggio di programmazione interpretato estensibile che combina una notevole potenza con una sintassi molto chiara.
Questa è la versione 0.9 (la prima versione beta), livello di patch 1.
Python può essere usato al posto degli script shell, Awk o Perl,...
...
Per favore provalo e mandami i tuoi commenti...
...
Sono l'autore di Python:
Guido van Rossum
CWI, dipartimento CST
Kruislaan 413
1098 SJ Amsterdam
Paesi Bassi
E-mail: gu...@cwi.ne
...
La fonte Python è protetta da copyright, ma è possibile utilizzarla e copiarla liberamente purché non si modifichi o si rimuova il copyright.»

Nasceva così, in pieno stile software libero, un linguaggio di programmazione che oggi domina nel campo della ricerca scientifica e non solo.

E' con un messaggio dello stesso tipo che, qualche mese più tardi, nell'agosto del 1991, Linus Torvalds lanciò quello che divenne il sistema operativo GNU/Linux.

Python e Linux, due bei coetanei: il pitone e il pinguino.

Mentre il pinguino pare sia stato scelto in quanto animale che godeva della simpatia di Linus Torvalds, il pitone proviene in realtà dal nome con cui Guido van Rossum battezzò il suo linguaggio in onore del gruppo di comici inglesi Monty Python, nei cui confronti egli nutriva sconfinata ammirazione.

In questo manualetto mi propongo di esporre le basi del linguaggio ad uso di coloro che vogliano provare a cimentarsi con la produzione di piccoli programmi per poi decidere se valga la pena approfondire per poter fare cose più impegnative.

La versione del linguaggio cui faccio riferimento è la 3, che ha ormai soppiantato la precedente versione 2.

Peraltro tutti i vari moduli di arricchimento del linguaggio sono ormai interamente disponibili anche per la versione 3.

A chi desideri conoscere la differenza tra le due versioni, magari per adattare alla versione 3 programmi scritti nella versione 2, suggerisco il mio articolo «Da Python 2 a Python 3» del giugno 2015, archiviato in Programmazione sul mio blog all'indirizzo *www.vittal.it*.

Indice

1 Installazione

Python si trova all'indirizzo *www.python.org*.

Sistema Linux

Il pacchetto base di Python è da sempre preinstallato sui sistemi operativi Linux.

La versione installata è l'ultima disponibile al momento dell'uscita della distro: in Debian e famiglia (Ubuntu e derivate, Mint) sono in genere preinstallate entrambe le versioni 2 e 3.

Se la o le versioni installate non ci aggradano più con il passare del tempo e non vogliamo aggiornare il sistema operativo, possiamo procurarci il source del pacchetto che vogliamo installare dal sito e installarlo, dopo averlo decompresso, con il solito procedimento

```
./configure
make
make test
sudo make install.
```

Sistema Windows

Windows non preinstalla Python e dobbiamo procurarci l'installer nella sezione Download del sito.

Sistema Mac OS X

Probabilmente vi troviamo installata la versione 2 ma, scrivendo a terminale il comando `python3` veniamo guidati ad installarlo.

* * *

Per lavorare bene con Python è consigliabile installare la IDLE (Integrated Development and Learning Environment).

Su Windows l'installazione avviene automaticamente quando lanciamo l'installer di Python. Su Linux e Mac OS X dobbiamo farlo noi ricorrendo al repository della nostra distro, nel caso di Linux, o, nel caso di Mac OS X all'indirizzo *https://www.python.org/download/mac/tcltk/*.

L'utilità di installare l'IDLE sta innanzi tutto nel fatto che, facendolo, automaticamente arricchiamo il pacchetto base del modulo tkinter, che ci consente di sviluppare in maniera abbastanza semplice applicazioni con interfaccia grafica.

A parte questa utilità l'IDLE ci offre una shell di Python con simpatiche e utilissime prestazioni di auto-inserimento di funzioni e parametri molto utile per l'apprendimento del linguaggio.

La IDLE contiene pure un editor per scrivere i programmi, assistiti dalle stesse prestazioni di auto-inserimento di funzioni e parametri.

2 Come funziona

Python è un linguaggio interpretato e interattivo.

Interpretato significa che il codice che scriviamo viene eseguito direttamente senza essere compilato in linguaggio macchina, come avviene con altri linguaggi (C, Pascal).

Interattivo significa che possiamo anche vedere eseguite le nostre istruzioni intanto che le scriviamo.

Quello che abbiamo installato secondo quanto indicato nel precedente Capitolo è l'interprete e viene avviato con il comando a terminale `python` (se abbiamo installato anche Python 2 il comando per l'interprete di Python 3 è `python3`)[1].

Se ci proponiamo di realizzare il nostro primo programma Ciao mondo possiamo sperimentarlo in due modi.

Possiamo scrivere in un file di testo, con un qualsiasi editor di testo, questa istruzione

```
print('Ciao mondo')
```

e salvare il file con estensione `.py`: ciaomondo.py.
Successivamente, e tutte le volte che vogliamo, posizionandoci nella directory dove abbiamo salvato il file e scrivendo a terminale

`python ciaomondo.py` (o `python3 ciamondo.py`)

vedremo eseguito il nostro programma (meglio chiamato script) che produrrà la voluta scritta.

L'altro modo è quello di avviare l'interprete scrivendo a terminale il comando

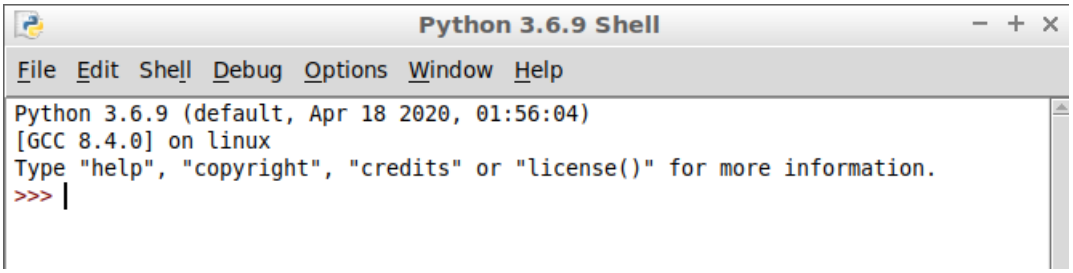
`python` (o `python 3`)

col che il terminale si trasforma in terminale Python, la shell Python, una conchiglia all'interno della quale si usa il linguaggio Python, e vi possiamo scrivere l'istruzione

```
print('Ciao mondo')
```

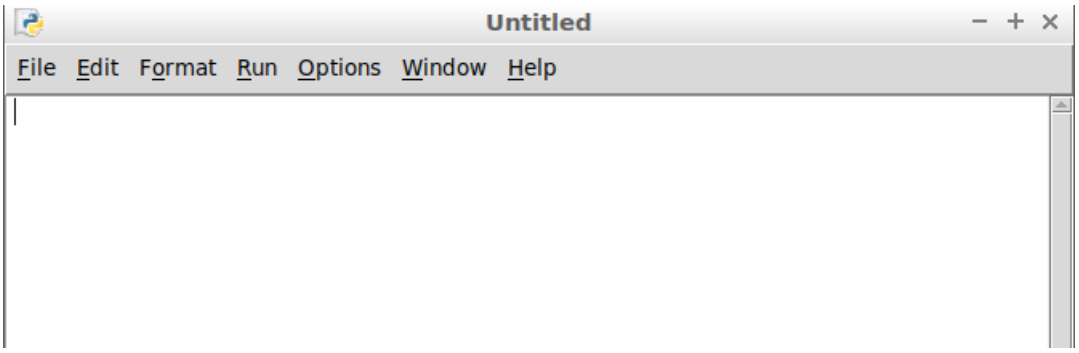
vedendola immediatamente eseguita alla pressione del tasto INVIO.

Se usiamo la IDLE, quando la lanciamo ci troviamo di fronte questa finestra



che è la shell di Python, nella quale ci possiamo divertire con l'interattività di Python.

Da menu FILE ▷ NEW FILE apriamo invece l'editor



nel quale possiamo scrivere il nostro programma, salvarlo con FILE ▷ SAVE AS, e lanciarlo con RUN ▷ RUN MODULE.

3 Tipi e tipizzazione

In Python, oltre ai soliti tipi che troviamo in tutti i linguaggi (numeri, stringhe) abbiamo anche tipi molto speciali che possono diventare utili in programmi di una certa complessità (liste, tuple, dizionari).

[1] Il terminale è una finestrella nella quale possiamo scrivere istruzioni con cui interagire con il computer. In Linux e Mac OS X si chiama proprio Terminale e troviamo modo di avviarlo dai menu a disposizione. In Windows si chiama Prompt dei comandi e si avvia con il comando `cmd` o `command`.

3.1 Numeri

Python tratta numeri interi, numeri in virgola mobile (dotati del separatore decimale `.`) e numeri complessi.

Per la dimensione di un numero intero (`int`) non c'è limite, salvo quello fisico della memoria del computer.

Per i numeri in virgola mobile (`float`) abbiamo la solita precisione limitata a 17 cifre, virgola compresa.

Nei numeri complessi (`complex`) parte reale e parte immaginaria, dotate di segno, sono scritte di seguito e la parte immaginaria è contrassegnata dal suffisso `j`.

Esempi:
`728` è un numero intero,
`7.5` è un numero in virgola mobile,
`-1+0.78j` è un numero complesso.

3.2 Stringhe

La stringa (`str`) è una sequenza di caratteri racchiusa tra apici semplici (`'`) o doppi (`"`). Se la racchiudiamo tra apici doppi, all'interno possiamo collocare apici singoli senza che questi la chiudano.

La stringa è immutabile, cioè non è possibile aggiungervi altri caratteri o toglierne.

Esempi:
`'Pippo'` è una stringa,
`"l'altro giorno"` è una stringa.

3.3 Tuple

La tupla (`tuple`) è una sequenza di elementi eterogenei, racchiusi tra parentesi tonde e separati da una virgola `,`).

La tupla è immutabile, cioè non è possibile aggiungervi altri elementi o toglierne.

Esempio:
`(1, 24, 'giuseppe')` è una tupla.

3.4 Liste

La lista (`list`) è una sequenza di elementi eterogenei, racchiusi tra parentesi quadre e separati da una virgola `,`).

La lista può essere modificata, cioè è possibile aggiungervi altri elementi e toglierne quando si vuole.

Esempio:
`[15, 'Ciao', (13, "Vittorio"), [16, 'Elena']]` è una lista.
Come si vede la lista, come la tupla, può contenere altre liste e tuple.

3.5 Dizionari

Il dizionario (`dict`) è una sequenza di elementi eterogenei, racchiusi tra parentesi graffe, contraddistinti da una chiave accoppiata a ciascun elemento.

Chiave e relativo elemento sono separati da due punti (`:`) e gli accoppiamenti sono separati da virgola (`,`).

Esempio:
`{'a': 12, 4: (22, 'pippo'), 'c':'Giuseppe'}` è un dizionario in cui `'a'`, `4` e `'c'` sono le chiavi.

3.6 Insiemi

L'insieme (`set`) viene creato dalla parola chiave `set` seguita, tra parentesi tonde, da una qualsiasi sequenza di elementi. Gli elementi dell'insieme sono tutti diversi uno dall'altro.

Esempi:
`set('Vittorio')` crea l'insieme delle lettere che compongono la parola Vittorio, scartando le doppie, con il risultato `{'V', 'r', 't', 'i', 'o'}`.

3.7 Valori booleani

I valori booleani (`bool`) sono `True` (vero) e `False` (falso).

3.8 Nulla

Il nulla (`NoneType`) è identificato dalla parola `None`.

* * *

Come scriviamo qualche cosa, Python, basandosi su ciò che scriviamo e su come lo scriviamo, assegna uno dei suoi tipi a ciò che scriviamo.

Esiste anche una funzione, la funzione `type()`, attraverso la quale possiamo verificare quale tipo Python assegna alla cosa che scriviamo.

Per esempio, se scriviamo nella shell di Python `type(122)`, viene ritornato <class 'int'>, a dimostrazione del fatto che Python ha capito che abbiamo scritto un numero intero.
Se scriviamo `type(True)` viene ritornato `<class 'bool'>`, a dimostrazione del fatto che Python ha capito che abbiamo scritto un valore booleano.
Se scriviamo `type([1, 4, 'ciao'])` viene ritornato `<class 'list'>`, a dimostrazione del fatto che Python ha capito che abbiamo scritto una lista.

4 Variabili

Le variabili sono delle locazioni di memoria, delle scatole, alle quali diamo un nome, destinate a contenere valori di un certo tipo.

In Python le variabili si creano nel momento in cui servono, senza bisogno, come avviene in quasi tutti gli altri linguaggi di programmazione, di dichiararle prima.

La tipizzazione della variabile è di tipo dinamico ed avviene automaticamente al momento dell'assegnazione del valore, in quanto Python, come abbiamo visto alla fine del precedente Capitolo, riconosce il tipo da come scriviamo il valore stesso.

L'assegnazione del valore viene fatta con l'operatore `=` e la sintassi
`<nome_variabile> = <valore>`.
`nome = 'Vittorio'` crea la variabile `nome` e le assegna il valore di tipo stringa `Vittorio`
`raggio = 6.5` crea la variabile `raggio` e le assegna il valore di tipo float `6,5`
`l = [15, 22, 8.5]` crea la variabile `l` e le assegna la lista `[15, 22, 8.5]`.

Il valore può essere espresso in modo letterale, come negli esempi, o in forma di espressione matematica, più o meno coinvolgendo altre variabili.

Il fatto che Python utilizzi la tipizzazione dinamica non vuol dire che sia un linguaggio che non dia importanza ai tipi: al contrario, Python è un linguaggio fortemente tipizzato.
Per esempio non è possibile sommare un valore numerico ad un valore di tipo stringa e non è possibile fare con una tupla ciò che si può fare con una lista.

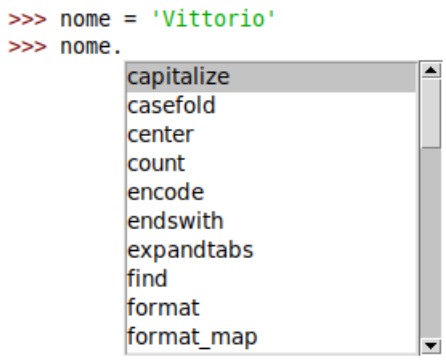
Infatti, a seconda del tipo assunto da una variabile, Python associa all'oggetto variabile tutta una serie di proprietà e metodi.

Esiste un comando, `dir`, che elenca proprietà e metodi di un oggetto passato come argomento con la sintassi
`dir(<argomento>)`

dove `<argomento>` può essere qualsiasi oggetto riconosciuto da Python (un modulo, una variabile, un oggetto creato da noi).

Meglio ancora, utilizzando la IDLE, possiamo vedere e scegliere i metodi associati ad una variabile scrivendo il nome della variabile stessa seguito da un punto (`.`) e attendere fino a quando ne vedremo l'elenco.

In questa illustrazione vediamo tutto ciò con riferimento ad una variabile stringa appena creata



Scorrendo l'elenco possiamo scegliere il metodo che ci interessa ed agire.
Se, per esempio, volessimo convertire la nostra stringa in caratteri maiuscoli, scorrendo l'elenco dovremmo scegliere il metodo `upper` e passare l'istruzione `nome.upper()`.
In questo modo produrremmo la stringa `'VITTORIO'` lasciando inalterata la stringa originaria contenuta nella variabile nome.
Se l'intenzione è di sostituirla l'istruzione deve essere `nome = nome.upper()`.
Se l'intenzione è di avere una nuova variabile con le lettere maiuscole accanto a quella con le lettere minuscole possiamo usare l'espressione `nome_maiuscolo = nome.upper()`.

Nell'economia di questo manualetto non posso certamente illustrare tutti i metodi che Python associa alle variabili nei vari tipi, anche perché probabilmente sconfinerei dall'intenzione di presentare qui le basi del linguaggio.

Utilizzando la IDLE, che non a caso sta per Integrated Development and Learning Environment, possiamo facilmente sperimentare i vari metodi che ci vengono offerti e stupirci di fronte alla ricchezza ed alla potenza di Python.

Può accadere di dover modificare il tipo di una variabile, per esempio per poter applicare ad essa metodi che non sarebbero propri del tipo assegnato dinamicamente al momento della sua creazione.

Abbiamo a disposizione le funzioni `int`, `float`, `str`, `list` e `tuple` per fare alcune trasformazioni.
Esempi:

sia `l` una variabile di tipo lista `[1, 6, 8]`
con `tuple(l)` la leggiamo come tupla e con `t = tuple(l)` la trasformiamo nella tupla `t`
in ogni caso la variabile `l` rimane tale e quale

sia `x` una variabile numerica a virgola mobile `13.7`
con `str(x)` la leggiamo come stringa `'13.7'` e con `s = str(x)` la trasformiamo nella stringa `s`
`con float(s)` rileggiamo la `s` come variabile numerica a virgola mobile
con `int(x)` leggiamo la parte intera di `x` e con `i = int(x)` trasformiamo la `x` nell'intero `i` (perdendo la parte decimale)
in ogni caso la variabile `x` rimane tale e quale

sia `st` una variabile stringa contenente il valore `'pippo'`
con `list(st)` la leggiamo come lista ['p', 'i', 'p', 'p', 'o'] e con `li = list(st)` la trasformiamo nella lista `li`
in ogni caso la variabile `st` rimane tale e quale

5 Operatori

Gli operatori collegano tra loro operandi di varia natura in espressioni che forniscono un risultato.

5.1 Operatori aritmetici

In ordine di esecuzione sono i seguenti:
`**` per l'elevamento a potenza tra numeri,
`*` per la moltiplicazione tra numeri,
`/` per la divisione tra numeri,
`%` per il modulo (resto della divisione intera),
`+` per la somma tra numeri o il concatenamento di stringhe,
`-` per la sottrazione tra numeri.

5.2 Operatori di confronto

Servono per confrontare due valori e il risultato che restituiscono è un valore booleano. Sono i seguenti:
`==` uguale,
`!=` non uguale,
`<` minore,
`<=` minore o uguale,
`>` maggiore,
`>=` maggiore o uguale.
Gli operandi assoggettati al confronto devono avere lo stesso tipo.

5.3 Operatori logici

Forniscono come risultato un valore booleano e sono i seguenti:
`&` per l'AND logico,
`|` per l'OR logico.

5.4 Operatori sugli insiemi

Agiscono tra operandi di tipo `set` e sono:
`|` per l'unione,
`-` per la differenza,
`&` per l'intersezione,
`^` per la differenza simmetrica (elementi presenti in uno dei due insiemi ma non in entrambi).

6 Interattività con l'utente

L'interfacciamento con l'utente nel terminale avviene attraverso due funzioni: `print` e `input`.

6.1 Output

La funzione per l'output è `print` con la sintassi
`print(`<cosa_scrivere>`)`
dove <cosa_scrivere> può essere indicato con una stringa, una espressione matematica o il nome di una variabile che contiene il valore che vogliamo scrivere, anche combinati tra loro separati con una virgola (`,`).

Esempi:
`print('ciao')` scrive `ciao`,

`print(5)` scrive `5`,
`print(3*7.5)` scrive `22.5`,
`print('3 per 8 fa', 3*8)` scrive `3 per 8 fa 24`,
data la variabile x contenente il valore `16`,
`print(x)` scrive `16`,
`print('la variabile x vale', x)` scrive `la variabile x vale 16`.

Con la direttiva di formattazione
`'%.<cifre_decimali>f' %`
possiamo stabilire le cifre decimali da scrivere arrotondando l'ultima.

Per esempio:
`print(7.168 * 4.68)` scrive `33.54624`
`print('%.2f' %(7.168*4.68))` scrive `33.55`
data la variabile `p` contenente il valore `3.141592653589793`
`print('%.5f' %p)` scrive `3.14159`

6.2 Input

La funzione per l'input è `input` con la seguente sintassi
`input(<eventuale_stringa_messaggio>)`
dove <eventuale_stringa_messaggio> può servire per chiedere all'utente che cosa inserire.

L'esecuzione del programma resta sospesa fino a quando l'utente non ha inserito da tastiera il dato richiesto.

Il dato inserito viene letto come stringa.

Se deve essere letto come numero il dato letto va reso argomento della funzione `eval()`.

Esempi:
`print(2**eval(input('esponente di due ')))` se inseriamo `3` scrive `8`
`raggio = eval(input('raggio '))` inserisce un numero nella variabile `raggio`
`x = input('Digita un numero ')` inserisce il numero digitato nella variabile `x` come stringa (se la variabile `x` deve entrare in un calcolo numerico dobbiamo preventivamente modificarne il tipo con `float(x)`).

7 Istruzioni complesse e indentazione

Capita spesso, soprattutto in programmi non banali, di dover gestire istruzioni in blocco e, in altri linguaggi di programmazione, queste istruzioni vengono in genere racchiuse tra parentesi graffe.

Python risolve il problema in maniera del tutto originale, attraverso l'indentazione, con la sintassi

```
<istruzione_apertura_blocco>:
    <istruzione>
    <istruzione>
    ....
<seguito_oltre_blocco>
```

La prima istruzione apre il blocco e termina con due punti (`:`) e il blocco è costituito dalle istruzioni rientranti sotto questa istruzione.

Come si esce dall'indentazione il blocco finisce.

8 Strutture di controllo

Come ogni altro linguaggio di programmazione, Python ha dei comandi per condizionare l'esecuzione di certe istruzioni al verificarsi di determinate condizioni oppure per la ripetizione dell'esecuzione di una o più istruzioni.

8.1 Esecuzione condizionale

if

L'istruzione `if`, chiamata istruzione di esecuzione condizionale (in inglese if è il nostro se), ci dà modo di assoggettare l'esecuzione di un blocco di istruzioni al verificarsi di una determinata condizione: se la condizione è vera viene eseguito il blocco di istruzioni, altrimenti si prosegue l'esecuzione del programma saltando il blocco stesso.

La sintassi è la seguente

`if` <condizione>:
    <istruzioni>

dove <condizione> è una qualsiasi espressione che relaziona due valori attraverso operatori di confronto: se la condizione si verifica vengono eseguite la o le istruzioni indicate, altrimenti si passa oltre.

L'istruzione `if` si presta anche all'esecuzione condizionale a due rami. Per ottenere questo dobbiamo abbinarla all'istruzione `else` con questa sintassi

`if` <condizione>:
    <istruzioni>
`else:`
    <istruzioni>

In questo caso se la condizione si verifica vengono eseguite le istruzioni contenute nel primo blocco altrimenti vengono eseguite quelle contenute nel secondo blocco dopo else (altrimenti). In ogni caso proseguendo poi nell'esecuzione del resto del programma.

Possiamo infine gestire l'esecuzione condizionale a più rami abbinando a `if` l'istruzione `elif` con questa sintassi

```
if <condizione>:
   <istruzioni>
elif <condizione>:
   <istruzioni>
elif <condizione>:
   <istruzioni>
else:
   <istruzioni>
```

8.2 Ripetizione

for

Si usa quando si vuole ripetere l'esecuzione di una istruzione o di un blocco di istruzioni per un numero definito di volte.

La sintassi per l'uso di questa istruzione in Python è del tutto originale e varia.

La situazione più semplice è quella che utilizza una sorta di contatore che è il costrutto `range`.

```
for i in range(n):
   <istruzioni>
```

esegue <istruzioni> `n` volte.

Numero delle iterazioni ed elementi su cui eseguirle possono essere determinati utilizzando un qualsiasi oggetto che sia una sequenza.

Per esempio:

```
for x in 'pippo':
   print(x)
```

elenca su cinque righe le lettere che compongono la stringa `'pippo'`

```
for x in 'pippo':
   print('ciao')
```

scrive cinque volte (quante sono le lettere della stringa `'pippo'`) la parola `ciao`.

while

Si usa per ripetere istruzioni fino a quando si verifica una certa condizione.

La sintassi è:

```
while <condizione>:
   <istruzioni>
```

Se la condizione è espressa attraverso l'uso di un contatore otteniamo gli stessi risultati che otteniamo con l'istruzione `for` vista prima

```
i = 1
while i <= 5:
   print "ciao"
   i = i + 1
```

scrive cinque volte la parola `ciao`.

L'istruzione `while`, combinata con `break`, si presta ad interrompere un ciclo al verificarsi di un certo evento, come l'inserimento di un certa stringa da tastiera:

con

```
while 1:
   x = input()
   if x == 'fine':
       break
```

essendo il valore 1 sempre vero, viene richiesto un input fino a quando non si scrive la parola `fine`.

9 Semplici programmi

Quanto visto finora ci consente di creare i nostri primi semplici script.

Questo calcola media, varianza e scarto quadratico medio di una serie di numeri:

```
lista = []
sx = 0
qx = 0
n = 0
print('Inserisci i dati (f per finire)')
while 1:
    xs = input()
    if xs == 'fine':
        break
    x = float(xs)
    lista.append(x)
    sx = sx + x
    qx = qx + x * x
    n = n + 1
m = sx / n
v = qx / n - m * m
sqm = v ** (1/2)
print("dati inseriti:")
for i in lista:
    print(i)
print('media:  ', '%0.4f' %m)
print('varianza:  ', '%0.4f' %v)
print('scarto quadratico medio:  ', '%0.4f' %sqm)
```

Notare che le variabili `lista`, `sx`, `qx` e `n` sono inizializzate al valore 0 prima del ciclo in quanto nel ciclo vengono lavorate come già esistenti (`lista.append`, `sx = sx + x`, ecc.).

Non avendo ancora visto come si estrae una radice quadrata, quella della varianza per determinare lo scarto quadratico medio è calcolata elevando la varianza a 1/2.
I dati in output comprendono l'elencazione dei numeri inseriti e i risultati sono espressi arrotondando a quattro cifre decimali.

Quest'altro script rivolge un saluto personalizzato al nome inserito con la tastiera:

```
nome = input('Come ti chiami?  ')
print('Ciao,', nome, '!')
```

La seconda riga potrebbe anche essere

```
print('Ciao, ' + nome + '!')
```

utilizzando l'operatore di concatenamento di stringhe.

Il seguente calcola l'area di un triangolo di cui si chiedono base e altezza:

```
b = eval(input('base:  '))
h = eval(input('altezza:  '))
a = b * h / 2
print("L'area di un triangolo di base", b, 'e altezza', h, 'è:', a)
```

10 Funzioni

La funzione è una sezione del programma in cui è racchiusa una serie di istruzioni da eseguire e che vengono eseguite ogni volta che la funzione è chiamata.

Per semplificare la scrittura di un certo programma, potrebbe essere utile raggruppare alcune istruzioni all'interno del programma stesso, creando funzioni richiamabili, in modo da suddividere i compiti e da evitare di scrivere più volte le stesse cose.

La sintassi per definire una funzione è

```
def <nome_funzione> (<nome_parametro>, <nome_parametro>,...):
   <istruzioni>
```

dove <nome_funzione> è il nome che intendiamo dare alla funzione, ed è il nome che useremo per richiamarla, <nome_parametro> è il nome del o dei parametri (dati) da inserire quando la richiamiamo.
Le <istruzioni> sono quelle relative alle elaborazioni da effettuare sui parametri per ottenere il risultato.
Se la funzione deve restituire un valore numerico, l'ultima istruzione è `return` con indicazione del risultato.

La sintassi per chiamare una funzione ed eseguirla è

```
<nome_funzione> (<parametro>, <parametro>,...)
```

Esempi:
definita la seguente funzione

```
def saluta(nome):
   print('Ciao ' + nome)
```

con l'istruzione
`saluta('Giuseppe')` scriviamo `Ciao Giuseppe`.
definita la funzione

```
def area_triangolo(base, altezza):
   return base * altezza / 2
```

con l'istruzione
`area_triangolo(12, 6)` otteniamo il risultato `36`.

Come esercizio esemplificativo supponiamo di voler scrivere un programma che calcoli Combinazioni e Disposizioni, semplici senza ripetizione, di n numeri presi k a k.
Le formule che dobbiamo utilizzare sono
$C(n,k) = \frac{n!}{(n-k)!k!}$ per le combinazioni,
$D(n,k) = \frac{n!}{(n-k)!}$ per le disposizioni,
nelle quali ricorre spesso il calcolo del fattoriale.

Il fatto che la necessità di questo calcolo ricorra spesso e comporti una non semplice scrittura di istruzioni per effettuarlo suggerisce di isolare queste istruzioni in una funzione dedicata al calcolo del fattoriale in modo da poter richiamare queste istruzioni in blocco quando serve.
La funzione per calcolare il fattoriale può essere scritta così:

```
def fattoriale (n):
    if n < 2:
       return 1
    return n * fattoriale (n - 1)
```

E' un bell'esempio di formula ricorsiva (funzione che richiama sé stessa) che ci dimostra come il linguaggio Python supporti la ricorsività.
Tra poco scopriremo che, nel modulo `math`, Python dispone della funzione `factorial()` per calcolare il fattoriale, per cui ci saremmo potuti evitare questa fatica.
Il programma per calcolare combinazioni e disposizioni di n numeri presi k a k utilizzando la funzione scritta da noi diventa il seguente:

```
def fattoriale (n):
    if n < 2:
        return 1
    return n * fattoriale(n-1)
n = eval(input('numero elementi:  '))
k = eval(input('presi:  '))
c = fattoriale(n)/(fattoriale(n-k) * fattoriale(k))
d = fattoriale(n)/fattoriale(n-k)
print('combinazioni:  ', c)
print('disposizioni:  ', d)
```

11 Moduli

Praticamente un modulo è una raccolta di funzioni.

Tra i moduli che fanno parte della dotazione basica di Python mi soffermo su quello che offre la maggiore utilità per un principiante.

11.1 Math

Come tutti i moduli, per essere utilizzato va importato con una particolare istruzione, da scrivere prima di ogni altra nello script, e lo si può fare in tre modi:

. `import math`
. `import math as <abbreviazione>`
. `from math import *`

In ogni caso l'operazione ci mette a disposizione le funzioni racchiuse nel modulo.
Nel primo caso la funzione che ci interessa si richiama come funzione membro dell'oggetto `math` con la sintassi `math.<funzione>`.
Nel secondo caso la funzione che ci interessa si richiama come funzione membro dell'oggetto ribattezzato con la sintassi `<abbreviazione>.<funzione>`.
Nel terzo caso possiamo richiamare la funzione che ci interessa semplicemente nominandola.

In ogni caso, se si tratta di funzione, si devono mettere tra parentesi tonde i parametri richiesti.

Esempio:
tra le funzioni del modulo `math` c'è `sqrt()` per estrarre la radice quadrata.
Per estrarre la radice quadrata di 9:
. se abbiamo importato con `import math` scriviamo `math.sqrt(9)`,
. se abbiamo importato con `import math as m` scriviamo `m.sqrt(9)`,
. se abbiamo importato con `from math import *` scriviamo `sqrt(9)`.

Il modulo `math` ci offre innanzi tutto i valori delle due più famose costanti e e π , rispettivamente richiamabili con `e` e `pi`.

Tra le funzioni di uso più ricorrente rammento le seguenti.

algebriche

`exp()` ritorna una potenza del numero e
`log()` ritorna il logaritmo naturale di un numero
`log10()` ritorna il logaritmo in base 10 di un numero
`sqrt()` ritorna la radice quadrata di un numero
`pow(base, esponente)` ritorna base elevata a esponente
`factorial()` ritorna il fattoriale di un numero
`trunc()` ritorna la parte intera di un numero
`floor()` ritorna l'intero inferiore di un numero decimale
`ceil()` ritorna l'intero superiore di un numero decimale

Forse balza all'occhio la mancanza di una funzione che determini il valore assoluto di un numero. In realtà Python dispone della funzione `abs()` al di fuori dal modulo `math`.

trigonometriche

`sin()` ritorna il seno
`asin()` ritorna l'arcoseno
`cos()` ritorna il coseno
`acos()` ritorna l'arcocoseno
`tan()` ritorna la tangente
`atan()` ritorna l'arcotangente

Gli argomenti per le funzioni trigonometriche vanno espressi in radianti.

Utili le seguenti funzioni di trasformazione:
`degrees(radianti)` trasforma radianti in gradi
`radians(gradi)` trasforma gradi in radianti

iperboliche

`sinh()` ritorna il seno iperbolico
`asinh()` ritorna l'arcoseno iperbolico
`cosh()` ritorna il coseno iperbolico
`acosh()` ritorna l'arcocoseno iperbolico
`tanh()` ritorna la tangente iperbolico
`atanh()` ritorna l'arcotangente iperbolico

11.2 Ricchezza di Python

Quello che abbiamo visto è il modulo di supporto al calcolo scientifico che troviamo nella dotazione di base.

Un altro modulo molto utile che ci troviamo installato se abbiamo installato la IDLE è il modulo `tkinter`, che ci consente di sviluppare applicazioni dotate di interfaccia grafica (GUI). In allegato all' articolo «Grafica con Python» del maggio 2018 sul mio blog all'indirizzo *www.vittal.it* si trova il manualetto tkinter.pdf.

Sempre sul blog troviamo l'articolo «Python per tutti» del febbraio 2017 con allegato il documento mondo_python.pdf che è una panoramica su ciò che il mondo del software libero ha sviluppato attorno all'idea originaria di Guido van Rossum.

Per chi voglia approfondire altri aspetti richiamo anche i seguenti articoli che si trovano sullo stesso blog:
«Python su Android» del giugno 2015 con allegato sl4a.pdf

«Ancora grafica con Python» dell'ottobre 2018 con allegato grafica_interattiva_python.pdf
«Software libero per data scientists» dell'aprile 2019 con allegato python_anaconda.pdf
«Ancora Python su Android» del maggio 2019 con allegato pydroid.pdf
«Software libero per il calcolo simbolico» del settembre 2019 con allegato sympy.pdf
«La blockchain secondo Python» dell'ottobre 2019 con allegato blockchain_python.pdf

12 Oggetti

Ovviamente Python supporta la programmazione a oggetti.

Tanti più esperti di me raggiungibili in rete possono ricordare o spiegare al lettore che cosa si intende per programmazione a oggetti.

Io provo a farlo capire con un esempio.

Supponiamo di avere spesso bisogno di determinare area e superficie del cubo di cui ci è noto lo spigolo.

Se vogliamo evitare di scrivere tutte le volte le necessarie formule, anche perché non è detto che ce le ricordiamo (a volte si ha a che fare con formule che più difficilmente di quelle per calcolare la superficie di un cubo sono rimandabili a memoria) possiamo creare delle funzioni che le contengono, come abbiamo fatto nel precedente Capitolo 10 per calcolare il fattoriale, oppure possiamo ricorrere alla programmazione a oggetti. Se seguiamo quest'altra strada scriviamo una Classe attraverso la quale creare un oggetto Cubo, contenente funzioni (chiamate funzioni membro) attraverso le quali calcolare volume e superficie.

Come si scrive una funzione lo abbiamo visto. Nel caso del nostro esempio la funzione per calcolare il volume del cubo sarà scritta così:

```
def volume_cubo(spigolo):
   return spigolo ** 3
```

Per calcolare il volume di un cubo di spigolo 5, scriveremo

```
volume_cubo(5)
```

Nella programmazione per oggetti, la classe per creare il cubo, con le funzioni membro per calcolare superficie e volume, sarà

```
class Cubo:
   def __init__(self, spigolo):
       self.s = spigolo
   def superficie(self):
       self.superficie = self.s * self.s * 6
       return self.superficie
   def volume(self):
       self.volume = self.s * self.s * self.s
       return self.volume
```

Utilizzando questa classe possiamo costruire un oggetto cubo con un certo spigolo, ad esempio 5:

```
mioCubo = Cubo(5)
```

e richiamarne le funzioni membro per scrivere superficie e volume:

```
print(mioCubo.superficie())
print(mioCubo.volume())
```

Il primo metodo inserito nella classe, dal nome prestabilito `__init__`, è il metodo costruttore, il cui primo parametro, che si usa chiamare `self`, ma potrebbe essere battezzato a piacere, ci permette di accedere all'oggetto che si sta per creare e, usato negli altri metodi, di accedere all'oggetto creato. L'altro parametro per il costruttore è lo spigolo, l'unico che, in questo caso, riteniamo adeguato per qualificare il cubo da costruire in vista delle grandezze da calcolare.
Se avessimo a che fare con una piramide, i parametri dovrebbero almeno essere: lato della base, numero dei lati della base e altezza della piramide.

Dopo di che abbiamo le due funzioni membro per determinare superficie e volume del nostro cubo, che andremo a costruire indicandone lo spigolo.

Il codice della classe e quello del suo utilizzo andrebbero inclusi nello stesso script.

Se si preferisce memorizzare a parte il codice della classe, lo si salva in un file con estensione `.py`, nel nostro caso, per esempio, `Cubo.py` nella stessa directory in cui salveremo lo script che lo utilizzerà.
La prima riga di questo script sarà
`from Cubo import *`.

13 Lavorare con file

Esiste la funzione `open()` con la quale possiamo creare un oggetto `file` dotato di funzioni per scriverne, aggiornarne o leggerne il contenuto.

La sintassi è la seguente
`f = open(<nome_file>, <modalità>)`
dove `f` è un qualsiasi nome che diamo all'oggetto `file`,
`<nome_file>` è una stringa che indica il percorso al file,
`<modalità>` è una stringa che può assumere i seguenti valori:
`'w'` se vogliamo scrivere sul file, creandolo se non c'è o sostituendolo,
`'a'` se vogliamo aggiungere elementi ad un file esistente senza sostituirlo,
`'r'` se vogliamo leggere il file.
Le più importanti funzioni membro del nostro oggetto file sono
`.write(<stringa>)` per scrivere una stringa nel file (per andare a capo occorre terminare la stringa con `\n`),
`.read()` per leggere tutto il file come unica stringa,
`.readline()` per leggere una riga del file (una volta letta una riga ci si posiziona sulla riga successiva e occorre ripetere il comando per leggerla),
`.readlines()` per leggere tutto il file e inserirne il contenuto in una lista i cui elementi sono costituiti da una riga del file.

Per rendere operativo ciò che abbiamo fatto sul file e chiuderlo posizionandosi sul primo elemento occorre utilizzare il metodo
`.close()`

Esempi:
con `f = open('/home/vittorio/Documenti/prova', 'w')`
creo il file `prova` nella posizione indicata dal percorso,
con `f.write('Pippo\n')` scrivo la sua prima riga con il nome Pippo,
con `f.close()` chiudo il file, confermando quanto scritto in esso,
con `f = open('/home/vittorio/Documenti/prova', 'a')`
apro il file `prova` per aggiungere altri elementi
con `f.write('Pluto\nPaperino\n')`
aggiungo altre due righe,
con `f.close()` chiudo il file, confermando quanto aggiunto,
con `f = open('/home/vittorio/Documenti/prova', 'r')`
apro il file `prova` per leggerne il contenuto
con `f.readline()` leggo la prima riga e, se ripeto il comando subito dopo, leggo la seconda e così via
con `f.close()` chiudo il file e riposiziono tutto.
Se riapro il file in lettura,
con `f.readlines()` ottengo la lista
`['Pippo\n', 'Pluto\n', 'Paperino\n']`
con `f.read()` otterrei la stringa
`'Pippo\nPluto\nPaperino\n'`

Il metodo `write` può scrivere una sola stringa per volta, per cui se vogliamo scrivere più stringhe dobbiamo concatenarle e se vogliamo scrivere numeri dobbiamo lavorare di conversione di tipo.

Per esempio, con il comando

```
f.write('3 moltiplicato 2 fa ' + str(3 * 2) + '\n')
```

scriveremmo nel nostro file la riga

```
3 moltiplicato 2 fa 6
```

Quello qui presentato è il modo di lavorare con i file basico di Python. Se l'esigenza è quella di lavorare su file di grandi dimensioni, contenenti dati da elaborare, non è certo questa la strada da percorrere e Pyhton ci offre ben altro. Per questo rimando a quanto citato nel Capitolo 11, paragrafo 11.2 Ricchezza di Python.
Esiste, per esempio, il modulo `pandas` che ci consente di leggere file che contengono tabelle, dati separati da virgola (csv) su più colonne, potendo anche specificare le colonne da leggere.

14 Lavorare con database

Con il linguaggio Python possiamo collegarci a un database per eseguirvi operazioni di scrittura o di lettura utilizzando il linguaggio SQL.

Esistono moduli adatti per i più noti database: `sqlite3` per sqlite3, `MySQLdb` per MySQL, `psycopg2` per PostGreSQL, `pyodbc` per MySQL, PostGreSQL, Access, Oracle.
Si tratta di moduli che non fanno parte della dotazione di base ma che vanno installati, per esempio utilizzando `pip3` secondo quanto indicato nell'allegato «mondo_python.pdf» al mio articolo «Python per tutti» del febbraio 2017 che si trova sul mio blog all'indirizzo *www.vittal.it*.

Una volta importato il modulo con

```
import <nome_modulo>
```

creiamo una connessione con il database, che chiamiamo con un nome evocativo, per esempio, `db`, con

```
db = <nome_modulo>.connect("<database>")
```

dove `<database>` sta per percorso e nome del database, messi tra apici come stringa.

Poi, per lavorare sul database con cui siamo collegati, utilizzando il metodo `cursor()` della connessione creiamo un cursore, che chiamiamo, per esempio, con la sua iniziale `c`, con

```
c = db.cursor().
```

I principali metodi del cursore così creato ci consentono di eseguire sul database comandi SQL o di leggere i risultati di una query, in particolare:

```
c.execute("<comando_SQL>")
```

esegue un comando SQL indicato come stringa o come variabile stringa;

```
risultato = c.fetchall()
```

crea una lista, chiamata risultato, nella quale ogni elemento è una tupla che rappresenta la riga di una astratta tabella con i dati della query per riga e colonna, come estratti dal database su cui è stata fatta la ricerca, dati a loro volta rappresentati sotto forma di stringhe unicode.

Alla fine del nostro lavoro dobbiamo chiudere il database con `db.close()`.

Facciamo un esempio: ci proponiamo di estrarre il contenuto della tabella dei Paesi di un database «biblioteca» che contiene titoli di libri di autori italiani, inglesi, statunitensi e russi.
Lo script Python sarà il seguente:

```
import sqlite3
db = sqlite3.connect("/home/vittorio/Database/biblioteca")
c = db.cursor()
c.execute("select * from paesi")
risultato = c.fetchall()
print(risultato)
db.close()
```

L'esecuzione dello script fornisce il seguente risultato:

```
[(1, 'Italia'), (2, 'Gran Bretagna'), (3, 'USA'), (4, 'Russia')]
```

Se non ci piace questo modo grezzo di presentare il risultato e lo vogliamo esporre meglio possiamo intervenire inserendo, al posto dell'istruzione `print(risultato)`, il seguente blocchetto:

```
for i in range(len(risultato)):
   ri = risultato[i]
   r = str(ri[0]) + " " + ri[1]
   print r
```
nel quale ogni elemento della lista risultato viene preso come una tupla isolata, ogni elemento della tupla viene accostato all'altro come stringa, con in mezzo un piccolo spazio libero, e ogni stringa risultante viene stampata e vediamo quanto segue:

```
1 Italia
2 Gran Bretagna
3 USA
4 Russia
```