15. Applicazioni del linguaggio Python

Il linguaggio Python (si veda l'Appendice A) nasce ad Amsterdam nel 1989, dove il suo creatore Guido Van Rossum lavorava come ricercatore. Nei suoi pochi anni di vita, questo linguaggio si è diffuso in tutto il mondo molto rapidamente. Uno degli aspetti che hanno decretato il successo Python è la facilità di apprendimento: chiunque nell'arco di pochi giorni può imparare a usarlo e a scrivere le sue prime applicazioni. Da questo punto di vista gioca un ruolo fondamentale la struttura aperta del linguaggio, priva di dichiarazioni ridondanti ed estremamente simile a un linguaggio parlato.
Consideriamo le possibili applicazioni di Python nella modifica dei file .pdb. Poiché tali file di testo contengono una riga per ciascun atomo della macromolecola, è facile immaginare come questi superino facilmente le migliaia di righe. Inoltre, ogni riga contiene determinati campi, ciascuno caratterizzato da informazioni specifiche sull'atomo descritto (tipo di atomo, posizione all'interno dell'amminoacido cui appartiene, identificatore di catena, coordinate spaziali, e così via).
Modificare manualmente molti file di tali dimensioni (esperienza piuttosto comune quando si utilizzano approcci di bioinformatica strutturale) è assai laborioso; per questo motivo risulta conveniente utilizzare script (piccoli programmi) costruiti *ad hoc,* in grado di automatizzare le modifiche ai file originari.

Script n. 1

Gli script in Python iniziano sempre con una riga che indica la locazione dell'interprete Python sul sistema operativo che state utilizzando. Nello specifico, il programma dovrà iniziare con i due simboli «#!» (cancelletto e punto esclamativo) seguiti (senza alcuno spazio!) dalla directory in cui si trova l'interprete.
Per trovare questa directory, aprite una shell e usate il comando linux **which**, seguito dal parametro *python* e annotate la risposta: con questa informazione siete pronti a scrivere la vostra prima linea di codice Python.
Il prossimo passo consiste nel creare un file di testo con estensione .py (per esempio *prova.py*). Per far ciò potete usare, per esempio, l'editor **gedit** (**gedit** *prova.py*) e salvare il file. Una volta creato il file di testo, sarà necessario renderlo eseguibile, cioè concedergli i permessi di esecuzione (il procedimento è spiegato nell'esercitazione su Linux).
A questo punto iniziamo a scrivere la prima riga di codice:

```
#!/usr/bin/python          (a patto che il vostro Python si trovi in /usr/bin)
```

Il vostro primo programma sarà molto semplice: stamperà sulla shell una frase (la tipica frase «Hello world!»); quindi la seconda riga di codice sarà la seguente:

```
print "Hello world!"
```

L'istruzione print è di semplice interpretazione, ossia stampa la frase racchiusa tra le virgolette (non omettetele!). Dove deve essere stampata la frase? Potreste volerla stampare su un file, o più semplicemente sullo schermo.
A questo punto salvate il file, tornate alla shell e lanciate il vostro programma digitando *prova.py* . Osservate il risultato (se il programma non dovesse essere eseguito, provate con: **python** *prova.py*).

Script n. 2

Ogni programma compie in genere operazioni più o meno complesse e ha quindi bisogno di memoria. Da questo punto di vista (il «consumo» di memoria) giocano un ruolo fondamentale le **variabili**. Potete pensare

a una variabile come a un pezzetto (molto piccolo) di memoria RAM del computer, isolato dal resto e con un nome ben preciso. La variabile ha un nome, e quando vogliamo riferirci al suo contenuto basta usare quel nome. Ricordate che il nome di una variabile non può contenere spazi, quindi non è possibile per esempio creare una variabile e chiamarla *nome e cognome* (potete usare invece qualcosa di simile a *nome_e_cognome*). Analizziamo il seguente script (create un nuovo file di testo, *somma.py*, salvatelo e rendetelo eseguibile).

```
#!/usr/bin/python
a = 153
b = 12
print "La somma di a e b:",  a+b
```

Lanciando il programma *somma.py* si osserva che l'interprete Python tratta le variabili *a* e *b* come numeri senza che voi abbiate dovuto specificarlo a priori (e ciò nei linguaggi di programmazione non è assolutamente scontato). In Python l'operatore matematico "+" può essere utilizzato anche per concatenare due stringhe. Vediamo il prossimo *script* (*concatena.py*):

```
#!/usr/bin/python
a = "98765"
b = "4321"
print "La stringa concatenata di a e b:", a+b
```

Lanciate il programma. In questo caso le variabili *a* e *b* sono state trattate come semplici stringhe, poiché il loro contenuto è stato messo tra apici (è indifferente in Python usare apici singoli o doppi).

Script n. 3

Proviamo ora a scrivere un programma un po' più utile. Invece di assegnare a priori dei valori alle variabili *a* e *b*, vogliamo che sia il programma a chiederceli, per poi effettuare un'operazione matematica. Scriviamo il seguente programma (*divisioni.py*):

```
#!/usr/bin/python
a = float(raw_input('Valore di a:'))
b = float(raw_input('Valore di b:'))
print "a diviso b:", a/b
```

Salvate il programma e fate delle prove inserendo i numeri quando vi vengono chiesti. Provate inoltre a inserire delle lettere nel primo numero e a dare il valore 0 come secondo parametro. Quali risultati ottenete? Come potete spiegarli?
Vediamo ora il significato delle istruzioni date. Il comando **raw_input** stampa sul terminale la stringa inserita tra le parentesi (e posta tra le virgolette!) e attende una stringa di risposta. La stringa inserita dall'utente viene poi convertita in un numero dall'istruzione **float** (l'operazione che muta un tipo di dato in un altro è chiamata ***cast***).
Quando inseriamo uno 0 al denominatore della divisione, il programma stampa un messaggio di errore: *ZeroDivisionError*. Possiamo prevedere un errore di questo tipo nel nostro programma e comportarci di conseguenza. In questo caso, faremo lanciare al programma un'**eccezione** al suo comportamento normale. Nelle istruzioni che stiamo per scrivere, si noti che abbiamo introdotto un carattere di tabulazione (**tab**) nella quinta e settima riga: questa prassi, importantissima in Python, viene definita **indentazione** ed è assolutamente necessaria per isolare i blocchi di una particolare istruzione (provate a non indentare il codice per vedere cosa succede):

```
#!/usr/bin/python
a = float(raw_input('Valore di a:'))
b = float(raw_input('Valore di b:'))
try:
        print "a diviso b:", a/b
```

```
except ZeroDivisionError:
        print "Impossibile"
```

Il programma prova (**try**) a eseguire la divisione. Se ci troviamo nel caso di un'eccezione di divisione per 0 (*except ZeroDivisionError*), allora il programma stamperà la parola «Impossibile». In questo modo possiamo gestire facilmente eventuali errori che si verificano durante l'esecuzione del programma.

Script n. 4

Passiamo ora all'ultimo script (*modif.py*), in grado di leggere un file .pdb e scrivere in output solamente informazioni sui carboni alfa della proteina analizzata: tutte le altre righe del file .pdb originario (relative all'*header*, agli altri atomi della proteina, e così via) non verranno visualizzate. Per ottenere il file .pdb collegatevi alla Protein Data Bank e inserite il codice 1BJ4. Salvate il file nella vostra *home directory*.

```
#!/usr/bin/python
nome_file=raw_input("Inserire il nome di un file pdb:")
try:
        pdb_file=open(nome_file)
except:
        print "Impossibile aprire il file."
        exit()
for riga in pdb_file:
        if 'ATOM' in riga and 'CA' in riga:
                print riga
pdb_file.close ()
```

Nella quarta riga di codice viene aperto un file in modalità *lettura* (si tratta del nostro file .pdb). Per operare su di un file, sia in lettura sia in scrittura, è necessario aprirlo associandogli un puntatore (***handler***, che in inglese significa «maniglia»): in questo caso il nostro puntatore al file in lettura è stato chiamato arbitrariamente *pdb_file*. Al termine delle operazioni sul file, il puntatore dovrà essere chiuso (osservate l'ultima riga di codice); questa operazione implica la distruzione del puntatore al file, che ormai non serve più.
Nell'ottava riga di codice introduciamo il costrutto *for*, il cui blocco ha il seguente significato: per ogni (*for*) riga (la variabile *riga*) presente nel (*in*) puntatore al *file* (*pdb_file*) esegui le istruzioni seguenti (:). Segue un blocco (indentato!) di istruzioni, nel nostro caso l'istruzione *if*: se (*if*) le parole «ATOM» e «CA» sono presenti nella riga ('*ATOM*' *in riga and* '*CA*' *in riga*) esegui le istruzioni seguenti (:). Segue nuovamente un blocco di istruzioni, *print riga*. Ogni volta che una riga viene letta, il costrutto *if* controlla se in essa sono presenti le stringhe «ATOM» e «CA» e se la condizione è soddisfatta stampa la riga in output.