

Programmazione (scientifica) in python

Liceo Scientifico e Sportivo Statale "A. Tassoni"

05/03/2018

I moduli

Problema: negli esercizi avete creato una funzione `primo`, per determinare se un numero è primo oppure no. Se voglio riutilizzare la funzione in un altro codice, come faccio?

La risposta sta nei *moduli*, ovvero collezioni di funzioni ed oggetti organizzati in una serie di files `.py`:

I moduli

Problema: negli esercizi avete creato una funzione `primo`, per determinare se un numero è primo oppure no. Se voglio riutilizzare la funzione in un altro codice, come faccio?

La risposta sta nei *moduli*, ovvero collezioni di funzioni ed oggetti organizzati in una serie di files `.py`:

1. Dalla pagina di gestione dei notebook, selezionare in alto a destra `New->Text File`
2. Rinominare il file in `interi.py`: qui ci mettiamo le funzioni che utilizziamo per operare con i numeri interi. Nel momento in cui la estensione del file viene letta come `.py`, l'editor del notebook evidenzia la sintassi del linguaggio python
3. Copiare il codice della funzione `primo` (compresa la definizione) nel file `interi.py`.

I moduli

Come utilizzo il codice di un modulo in un notebook?

Bisogna *importare* i contenuti del modulo. Ci sono tre metodi principali:

Metodo 1

```
import interi    # <- qui non ci va l'estensione .py
x = 23
if interi.primo(x):
    print(x, "è un numero primo")
else:
    print(x, "è non un numero primo")
```

Le funzioni definite nel modulo le uso come *metodi* del modulo stesso

I moduli

Metodo 2

```
from interi import primo
x = 23
if primo(x):
    print(x, "è un numero primo")
else:
    print(x, "è non un numero primo")
```

I moduli

Metodo 2

```
from interi import primo
x = 23
if primo(x):
    print(x, "è un numero primo")
else:
    print(x, "è non un numero primo")
```

Metodo 3

```
from interi import * # <- con l'asterisco si importano
                    # TUTTI i contenuti del modulo
x = 23
if primo(x):
    print(x, "è un numero primo")
else:
    print(x, "è non un numero primo")
```

In questi due casi `primo` diventa una funzione definita *localmente*.

I moduli

Ma c'è un problema: e se ho una variabile nel mio codice che già si chiama `primo` (per esempio, se ci ho immagazzinato il primo numero primo, cioè 2)? Si ha il cosiddetto *naming conflict* (conflitto di denominazione)

I moduli

Ma c'è un problema: e se ho una variabile nel mio codice che già si chiama `primo` (per esempio, se ci ho immagazzinato il primo numero primo, cioè 2)? Si ha il cosiddetto *naming conflict* (conflitto di denominazione)

	Pro	Contro
Metodo 1	difficile che si creino conflitti di <i>denominazione</i>	ogni volta bisogna digitare il nome del modulo
Metodo 2	si risparmia tempo nello scrivere il codice	si possono generare conflitti di <i>denominazione</i>
Metodo 3	si risparmia tempo nello scrivere il codice	si possono generare conflitti di <i>denominazione</i>

Il metodo 3 è generalmente da evitare, in quanto se un modulo contiene molte funzioni si rischia di importare molte cose che non vengono utilizzate, con aumento di probabilità di "conflitti".

I moduli

Le soluzioni sono:

Metodo 1 con abbreviazione (sintassi **as**)

```
import interi as nt # <-nt è un "alias" (più breve) per il modulo originale  
print(nt.primo(29))
```

I moduli

Le soluzioni sono:

Metodo 1 con abbreviazione (sintassi `as`)

```
import interi as nt # <-nt è un "alias" (più breve) per il modulo originale  
print(nt.primo(29))
```

Metodo 2 con nomi a basso rischio di ambiguità

per esempio, rinominando `primo` in `determina_se_primo`

```
from interi import determina_se_primo  
print(determina_se_primo(29))
```

I moduli

Le soluzioni sono:

Metodo 1 con abbreviazione (sintassi `as`)

```
import interi as nt # <-nt è un "alias" (più breve) per il modulo originale
print(nt.primo(29))
```

Metodo 2 con nomi a basso rischio di ambiguità

per esempio, rinominando `prim` in `determina_se_primo`

```
from interi import determina_se_primo
print(determina_se_primo(29))
```

In ogni caso, occorre digitare qualche carattere in più.

Io preferisco la prima soluzione, in quanto mi consente di capire subito da quale modulo vengono le funzioni (nel caso che ci sia qualcosa che non va).

I moduli

Alcune raccomandazioni generali

- nei files `.py` che contengono le funzioni di un modulo, è bene mettere come prima riga la seguente:

```
from __future__ import print_function, division
```

Questo vi evita un sacco di idiosincrasie dovute al fatto che qualcuno ancora usa la versione 2 di Python (noi usiamo la 3, la più recente).

I moduli

Alcune raccomandazioni generali

- nei files `.py` che contengono le funzioni di un modulo, è bene mettere come prima riga la seguente:

```
from __future__ import print_function, division
```

Questo vi evita un sacco di idiosincrasie dovute al fatto che qualcuno ancora usa la versione 2 di Python (noi usiamo la 3, la più recente).

- i comandi `import` vanno sempre messi **all'inizio** dei notebook. Questo non è strettamente necessario, ma rende più leggibile il vostro lavoro.
- È buona prassi avere una sola cella contenente tutti i comandi `import`, e solo quelli.

I moduli

Alcune raccomandazioni generali

- nei files `.py` che contengono le funzioni di un modulo, è bene mettere come prima riga la seguente:

```
from __future__ import print_function, division
```

Questo vi evita un sacco di idiosincrasie dovute al fatto che qualcuno ancora usa la versione 2 di Python (noi usiamo la 3, la più recente).


- i comandi `import` vanno sempre messi **all'inizio** dei notebook. Questo non è strettamente necessario, ma rende più leggibile il vostro lavoro.
- È buona prassi avere una sola cella contenente tutti i comandi `import`, e solo quelli.
- All'inizio di ciascun modulo, così come all'inizio di ogni funzione, è bene inserire una *docstring* che descriva lo scopo dei contenuti del modulo

I moduli

Alcune raccomandazioni generali

- l'uso dei notebook comporta qualche fastidio quando si lavora con i moduli: se
 1. si scrive un modulo
 2. si importa in un notebook
 3. si modifica il modulo
 4. si importa nuovamente nel notebook

il contenuto del modulo il più delle volte non viene modificato correttamente.

Per questo motivo, tra il punto 3. e 4. bisogna prima far ripartire il *kernel* dell'interprete, utilizzando o il comando `Kernel->Restart` o il tasto  nella barra degli strumenti. Questo ha lo spiacevole effetto di resettare tutte le variabili, per cui occorre eseguire di nuovo tutte le celle di codice.

Alcuni moduli di particolare importanza

Python, come altri linguaggi, dispone di una *"biblioteca (libreria) standard"* di moduli contenenti funzioni molto comuni. ([link](#))

Ad esempio...

`math` ([link](#))

contiene le funzioni e costanti matematiche più utilizzate.

```
import math

print(math.pi)      # <- pi greco
print(math.log(14))  # <- logaritmo naturale
print(
    math.sin(
        math.radians(180)  # vuole l'argomento in radianti
    )
)
```


"Biblioteche" importanti(ssime)

la vera potenza di python sta nel fatto che è già stato scritto un modulo per quasi tutte le esigenze. In particolare (già presenti in Anaconda, se no vanno scaricate):

"Biblioteche" importanti(ssime)

la vera potenza di python sta nel fatto che è già stato scritto un modulo per quasi tutte le esigenze. In particolare (già presenti in Anaconda, se no vanno scaricate):

numpy

fornisce una serie di funzioni e utilità per lavorare con **vettori** (*array*)

```
import numpy as np  # <- di solito si abbrevia
```

```
print(np.array([2, 3, 5, 15, 63, 12]))    # <- tutti interi  
print(np.array([2, 3, 5, 15.2, 63, 12]))  # <- tutti reali  
print(np.array([2, 3, 5, 15, 63, "ascas"])) # <- tutte stringhe
```

"Biblioteche" importanti(ssime)

la vera potenza di python sta nel fatto che è già stato scritto un modulo per quasi tutte le esigenze. In particolare (già presenti in Anaconda, se no vanno scaricate):

numpy

fornisce una serie di funzioni e utilità per lavorare con **vettori** (*array*)

```
import numpy as np # <- di solito si abbrevia
```

```
print(np.array([2, 3, 5, 15, 63, 12])) # <- tutti interi  
print(np.array([2, 3, 5, 15.2, 63, 12])) # <- tutti reali  
print(np.array([2, 3, 5, 15, 63, "ascas"])) # <- tutte stringhe
```

gli elementi degli array **numpy** sono tutti dello stesso tipo.

numpy

ma non bastavano le liste?? no....

```
x = [1, 3]
y = [0, -4]
print(x+y)      # <- ma io li volevo sommare
print(x*y)      # <- qui vorrei moltiplicare
```

numpy

ma non bastavano le liste?? no....

```
x = [1, 3]
y = [0, -4]
print(x+y)      # <- ma io li volevo sommare
print(x*y)      # <- qui vorrei moltiplicare
```

le operazioni con le sequenze di valore si fanno con `numpy`

```
x = np.array([1, 3])
y = np.array([0, -4])
print(x+y)
print(x*y)
```

numpy

Qualche metodo interessante

```
x = np.array([1,2,3,4,5,6,7])  
print(x.sum())      # <- somma  
print(x.mean())     # <- media
```

numpy

Qualche metodo interessante

```
x = np.array([1,2,3,4,5,6,7])
print(x.sum())      # <- somma
print(x.mean())     # <- media
```

anche le sequenze di valori si possono generare velocemente:

```
print(np.arange(0,100,3))    # <- si usa come la funzione range
# oppure
print(np.linspace(0,100,10))
#
#           ↑
#           L-- il terzo argomento è il numero di "passi"
```

numpy

fornisce una estensione di tutte le funzioni del modulo `math` per operare su sequenze di valori:

```
x = np.array([0, 30, 45, 60, 90, 120, 150, 180])
print(np.sin(np.radians(x)))
y = np.array([1, 10, 100, 100])
print(np.log10(y))
```


matplotlib

è la libreria "principe" per disegnare grafici o visualizzare immagini. A noi servirà principalmente per "plottare" il risultato di alcuni conti.

per usare `matplotlib` in un notebook, nella *prima* cella del notebook digitare:

```
%matplotlib inline  
import matplotlib.pyplot as plt
```

matplotlib

è la libreria "principe" per disegnare grafici o visualizzare immagini. A noi servirà principalmente per "plottare" il risultato di alcuni conti.

per usare `matplotlib` in un notebook, nella *prima* cella del notebook digitare:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

supponiamo di voler riportare in un grafico i punti $A=(0,0)$, $B=(1,3)$, $C=(-3,2)$ e $D=(2,2)$.

```
x = np.array([0, 1, -3, 2]) # <- tutte le coordinate x
y = np.array([0, 3, 2, 2]) # <- tutte le coordinate y
```

matplotlib

è la libreria "principe" per disegnare grafici o visualizzare immagini. A noi servirà principalmente per "plottare" il risultato di alcuni conti.

per usare `matplotlib` in un notebook, nella *prima* cella del notebook digitare:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

supponiamo di voler riportare in un grafico i punti $A=(0,0)$, $B=(1,3)$, $C=(-3,2)$ e $D=(2,2)$.

```
x = np.array([0, 1, -3, 2]) # <- tutte le coordinate x
y = np.array([0, 3, 2, 2]) # <- tutte le coordinate y
```

```
plt.plot(x, y, 'o')
```

matplotlib

è la libreria "principe" per disegnare grafici o visualizzare immagini. A noi servirà principalmente per "plottare" il risultato di alcuni conti.

per usare `matplotlib` in un notebook, nella *prima* cella del notebook digitare:

```
%matplotlib inline
import matplotlib.pyplot as plt
```

supponiamo di voler riportare in un grafico i punti $A=(0,0)$, $B=(1,3)$, $C=(-3,2)$ e $D=(2,2)$.

```
x = np.array([0, 1, -3, 2]) # <- tutte le coordinate x
y = np.array([0, 3, 2, 2]) # <- tutte le coordinate y
```

```
plt.plot(x, y, 'o')
```

```
plt.plot(x, y) # <- senza il terzo argomento, collega i punti con
               # una spezzata
```

Chiudere tutti i notebook

