Matematica con Python

Mauro Saita

e-mail: maurosaita@tiscalinet.it

Versione provvisoria. Dicembre 2016.[1]

Lezione n. 3

	2	3	5	7	11	13	17	19	23
29	31	37	41	43	47	53	59	61	67
71	73	79	83	89	97	101	103	107	109
113	127	131	137	139	149	151	157	163	167
173	179	181	191	193	197	199	211	223	227
229	233	239	241	251	257	263	269	271	277
281	283	293	307	311	313	317	331	337	347
349	353	359	367	373	379	383	389	397	401
409	419	421	431	433	439	443	449	457	461
463	467	479	487	491	499	503	509	521	523
541	547	557	563	569	571	577	587	593	599
601	607	613	617	619	631	641	643	647	653
659	661	673	677	683	691	701	709	719	727
733	739	743	751	757	761	769	773	787	797
809	811	821	823	827	829	839	853	857	859
863	877	881	883	887	907	911	919	929	937
941	947	953	967	971	977	983	991	997	

Numeri primi fino a 1000.

Indice

[1] Nome File: python_lezione_03_2016.tex

1 Funzioni

Una funzione è un programma a cui è stato dato un nome in modo che possa essere chiamato, cioè eseguito, tramite quel nome. La sintassi per definire una funzione è

```
def nome_funzione(parametri):
   <istruzioni>
```

Per esempio, la seguente funzione, denominata 'divide', stabilisce se b è un divisore di a. L'output della funzione è prodotto tramite il costrutto *return*; in questo caso, se b divide a la funzione restituisce il valore 'True' altrimenti 'False'.

```
def divide(b, a):
         if a % b == 0:
            return True
         else:
            return False
```

Le funzioni sono sottoprogrammi che possono essere chiamati tutte le volte che lo si ritiene necessario. Il seguente programma acuisisce due interi da tastiera, controlla se il primo è un divisore del secondo e, in caso affermativo stampa: 'b è un divisore di a' altrimenti stampa: 'b è NON è un divisore di a'.

```
def divide(b, a):
         if a % b == 0:
            return True
         else:
            return False


# MAIN DEL PROGRAMMA
b = input('Digitare un intero positivo: b = ')
a = input('Digitare un altro intero positivo: a= ')
b = float(b)
a = float(a)
if divide(b, a)==True:
     print(int(b), è un divisore di ', int(a))
else:
     print(int(b), ' NON è un divisore di ', int(a))
```

1.1 Trasformazione di un intervallo di tempo da 'secondi' a 'ore, minuti, secondi'.

```
# Nome programma tempo01.py

# Main del programma
nsec = input('Inserire un tempo espresso in secondi: ')
nsec = int(nsec)
# Il quoziente hh tra n e 3600 fornisce il numero di ore.
hh = nsec//3600
# Il resto r di nsec e 3600 dà i secondi restanti
r  =  nsec % 3600
# Il quoziente mm tra r e 60 fornisce il numero di minuti.
mm = r // 60
# Il resto ss di r e 60 fornisce i secondi.
ss = r % 60

#Tempo in hh mm ss
print('Tempo espresso in ore, minuti, secondi: ')
print(str(hh) + ' hh ' + str(mm)+ ' mm ' + str(ss)+ ' ss ' )
```

Si può scrive lo stesso programma utilizzando una funzione (qui denominata 'hhmmss'):

```
# Nome programma tempo02.py

def hhmmss(nsec):
    # Il quoziente hh tra n e 3600 fornisce il numero di ore.
    hh = nsec//3600
    # Il resto r di nsec e 3600 dà i secondi restanti
    r = nsec % 3600
    # Il quoziente mm tra r_1 e 60 fornisce il numero di minuti.
    mm = r // 60
    # Il resto ss di r e 60 fornisce i secondi.
    ss = r % 60
    return str(hh)+ ' hh ' + str(mm)+ ' mm '  + str(ss) + ' ss '

# Main del programma
n = input('Inserire un tempo espresso in secondi: ')
n = int(n)
tempo=hhmmss(n)
print('Tempo espresso in ore, minuti, secondi: ' + tempo)
```

2 Esercizi

Esercizio 2.1.

(a) *Scrivere la funzione 'max2' che restituisce il massimo tra* 2 *numeri.*

(b) *Scrivere il programma che, usando la funzione 'max2' legge* 2 *numeri e stampa il maggiore.*

Esercizio 2.2. *Scrivere le funzioni 'min3' e 'max3' che restituiscono rispettivamente il minimo e il massimo tra* 3 *numeri*

Esercizio 2.3.

(a) *Scrivere la funzione 'ordina2' che restituisce due numeri in ordine crescente.*

(b) *Scrivere la funzione 'ordina3' che restituisce tre numeri in ordine crescente.*

Esercizio 2.4. *Scrivere una funzione che trova il più piccolo divisore di un intero* $n \neq 0, \pm 1$.

Esercizio 2.5. *Scrivere una funzione che calcola il numero di diagonali di un poligono di* n *lati.*

3 Numeri primi e teorema fondamentale dell'aritmetica

Se a e b sono due interi con $b > 0$ esistono sempre, e sono unici, il quoziente q e il resto r della divisione di a per b. Se il resto di tale divisione è zero ($r = 0$) si ha: $a = bq$ e b risulta essere un *divisore* di a. Naturalmente ogni intero positivo ha *almeno* due divisori: 1 e il numero stesso. Il numero 360 ha in tutto 24 divisori mentre 359 ne ha esattamente due (1 e 359). Si chiamano *numeri primi* gli interi positivi (maggiori di 1) che hanno esattamente due divisori; questi numeri rivestono un ruolo molto importante in diversi ambiti della matematica.

Definizione 3.1. *Un intero* $n > 1$ *si dice* primo *se i suoi unici divisori positivi sono* 1 *e* p*; si dice* composto *se non è primo.*

Un primo fatto che evidenzia l'importanza dei numeri primi è questo: ogni intero $n > 1$ si può sempre esprimere come prodotto di numeri primi; per esempio $2860 = 2^2 \cdot 5 \cdot 11 \cdot 13$.

Ogni intero n maggiore di 1, o è primo, e in tal caso la sua fattorizzazione coincide con il numero stesso, oppure è composto; in questo secondo caso $n = qd$, con q e d diversi da 1 e n. Se il numeri q e d non sono primi si possono a loro volta scrivere come prodotto di due interi (diversi da 1 e dal numero)... . Iterando più volte questo ragionamento si scrive n come prodotto di primi

$$n = p_1 \cdots p_h \tag{3.1}$$

gli interi p_i sono primi non necessariamente distinti.

Teorema 3.2. (Teorema fondamentale dell'aritmetica) *Ogni intero $n > 1$ può essere scritto come prodotto*

$$n = p_1 \cdots p_h$$

dove gli interi $p_1 \cdots p_h$ sono primi e $h \geq 1$. Questa espressione è unica, a meno dell'ordine dei fattori primi.

4 Test di primalità

Un test di primalità consiste in un metodo per stabilire se un dato numero, diciamo n, è primo oppure no. Un semplice algoritmo per risolvere questo problema è il seguente:

Algoritmo elementare per un test di primalità.
Sia n un intero positivo, con $n > 1$.
Si divida n per: $2, 3, 4, \ldots (n-1)$.

- Se nessuna divisione dà resto 0 allora n è primo.
- Se almeno una divisione dà resto 0 allora n è composto.

Esercizio 4.1.

(a) *Scrivere diagramma di flusso pseudocodifica dell'algoritmo riportato sopra.*

(b) *Tradurre la pseudocodifica in un programma (acquisito un intero $n > 1$ da tastiera, stabilire se quel numero è primo oppure no).*

*Denominare il programma: "**primalita01.py**"*

P.S. Il numero 1789, anno della Rivoluzione Francese, è primo.

5 Cercare risposte . . . in tempi ragionevoli!

Nella sezione precedente è stato realizzato un programma in grado di stabilire se un certo numero n è primo oppure no. Si tratta di un programma efficiente? Ossia, in quanto tempo

è in grado di fornire risposte alla nostre domande? Ovviamente i tempi di esecuzione variano a seconda della grandezza del numero ...

Esercizio 5.1.

(a) *Trovare il più grande numero primo di* 10 *cifre.*

(b) *Qual è il tempo di esecuzione del programma "primalita01.py" nel caso del numero primo trovato al punto precedente?*

Esercizio 5.2. ***[Tempi di esecuzione per il test di primalità]***
Costruire una tabella come quella qui sotto riportata. Per ogni numero

Numeri interi	*Tempi di esecuzione per test di primalità*
Interi di 8 *cifre*	...
Interi di 9 *cifre*	...
Interi di 10 *cifre*	...
Interi di 12 *cifre*	...
.....	.....

5.1 Analisi e ottimizzazione dell'algoritmo

Per effettuare il test di primalità per il numero $n = 1789$ (che è primo) il programma "primalita01.py" passa in rassegna tutti i numeri d da 1 a 1789 e per ognuno di essi stabilisce se esso è un divisore di n oppure no. In tutto deve effettuare

1789 iterazioni

Ovviamente, si può evitare di eseguire il test per $d = 1$ e $d = n$ (1 e n sono certamente divisori) e così le iterazioni necessarie sono 1787.

- Osservando che il più grande divisore di n (diverso da n) non può essere maggiore di $\frac{n}{2}$ si può limitare la ricerca di divisori nell'intervallo che va da 2 a $\frac{n}{2}$. In questo modo è sufficiente effettuare

893 iterazioni

- Il seguente teorema permette un ulteriore miglioramento

Teorema 5.3. *Sia $n \in \mathbb{Z}$, $n > 1$.*

Se n è composto allora ammette almeno un divisore proprio minore o uguale di $\sqrt{n}$.

Dimostrazione.

Per ipotesi n è composto, allora si può scrivere:

$$n = ab$$

con a, b interi maggiori di 1 e minori di n. I numeri a e b non possono essere entrambi maggiori o entrambi minori di $\sqrt{n}$, altrimenti il loro prodotto sarebbe maggiore oppure minore di n. Segue che n deve avere un divisore proprio compreso tra 2 e $\sqrt{n}$. ∎

Quindi è sufficiente ricercare un divisore di 1789 compreso tra 2 e $\sqrt{1789}$. In tutto sono sufficienti

41 iterazioni

- Un qualsiasi numero naturale n si può scrivere nella forma:

$$6k \quad 6k+1 \quad 6k+2 \quad 6k+3 \quad 6k+4 \quad 6k+5$$

con $k \in \mathbb{N}$. I numeri della forma $6k$, $6k+2$, $6k+4$ sono pari, quindi per i numeri di questo tipo basta verificare la divisibilità per 2, infatti se 2 è un divisore di n non serve cercare altri divisori, altrimenti (se 2 non è un divisore di n) ogni numero del tipo $6k$, $6k+2$, $6k+4$, non è un divisore di n.

I numeri della forma $6k+3$ sono multipli di 3, quindi per i numeri di questo tipo basta testare la divisibilità per 3 (se 3 è un divisore di n non serve cercare altri divisori, altrimenti ogni numero del tipo $6k+3$ non è un divisore di n).

Quindi, dopo aver testato la divisibilità per 2 e 3 si può limitare la ricerca di eventuali altri divisori tra i numeri del tipo $d = 6k+1$ e $d = 6k+5$.

In conclusione, per $n = 1789$ la ricerca dei suoi divisori può essere ridotta ai seguenti casi:

$d = 2$ e $d = 3$;

numeri del tipo: $d = 6k+1$, $7 \leq d < \sqrt{1789}$ cioè : $7, 13, 19, 25, 31, 37$;

numeri del tipo: $d = 6k+5$, $5 \leq d < \sqrt{1789}$ cioè , $5, 11, 17, 23, 29, 35, 41$.

Si tratta di eseguire in tutto

15 iterazioni

5.2 Il modulo time

Il modulo time contiene funzioni per tempo e data. Ecco alcuni esempi:

```
import time

print(time.localtime())
#output: time.struct_time(tm_year=2016, tm_mon=12, tm_mday=7, tm_hour=17,
# tm_min=7, tm_sec=4, tm_wday=2, tm_yday=342, tm_isdst=0)

print (time.time())
#output: 1481127169.915939

print (time.strftime('%d %b %Y, %H:%M',t))
#output: 07 Dec 2016, 17:07

t=time.ctime()
print(time.ctime())
#output: Wed Dec  7 17:21:00 2016

print(time.clock())
#output: 8.210962620413767e-07

t=time.clock()
t=format(t, '10.6f')
print(t)
#output: 227.492659
```

Esercizio 5.4.

(a) *Modificare il programma "primalita01.py" tenendo conto delle osservazioni della sezione precedente e denominare la nuova versione "primalita02.py".*

(b) *Calcolare i tempi di esecuzione della nuova versione del programma ("primalita02.py") nei casi già analizzati nell'Esercizio 5.2. Confrontare i risultati.*