Funzioni - Parte 2

Mirko Cesarini - Dario Pescini
nome.cognome@unimib.it

Università di Milano Bicocca

Ripasso: parametri di una funzione

- Parametri
  - Formali
  - Attuali
- Definizione di funzione

```
1  def moltiplica(x, y):   #x ed y sono parametri formali
2      return x * y        # return comunica il risultato al
3                          # programma chiamante e termina
4                          # l'esecuzione della funzione
5  #Uso della funzione
6  a=10
7  b=moltiplica(a,20)   # a e 20 sono parametri attuali
8  print(b)
```

```
200
```

Modalità di passaggio dei parametri

- Quando si richiama una funzione, vengono trasferiti i valori dai *parametri attuali* ai *parametri formali* della funzione
- Il passaggio di parametri può avvenire in due modalità distinte:
  - Passaggio per *valore* (o per copia): il parametro formale è una copia del parametro attuale se la funzione modifica il contenuto del parametro formale, si modifica la copia e non l'originale
  - Passaggio per *riferimento*: il parametro formale è collegato direttamente al parametro attuale modificando il valore del parametro formale si altera anche il valore del parametro attuale
- La differenza si nota quando, all'interno di una funzione, viene modificato un parametro formale
  - (nel passaggio per valore) il corrispondente parametro attuale rimane invariato
  - (nel passaggio per riferimento) il corrispondente parametro attuale viene modificato

Tipi delle variabili e passaggio di parametri

- In Python i **tipi semplici** (int, float, . . .) sono passati per valore (le funzioni . . . lavorano su una copia)
- I **tipi complessi** (liste, dizionari, . . .) sono invece passati per riferimento (le funzioni . . . lavorano sull'originale)
- Esiste una categoria intermedia: i **tipi complessi non mutabili** (es. stringhe, tuple).
  - Non possono essere modificati . . .
  - Non si pone il problema del passaggio per copia o per valore
  - (Per la cronaca) sono passati per riferimento

Passaggio di parametri: esempio 2

- Funzione swap non funzionante

```
1  def swap (x, y): # x ed y sono passati per copia
2      temp = x
3      x=y
4      y=temp
5
6  a=100
7  b=30
8  swap(a, b)
9  print ("a: %d, b: %d" % (a,b))
```

```
a: 100, b: 30
```

Swap funzionante

```
def swap (g):
    temp = g[0]
    g[0]=g[1]
    g[1]=temp
p=[100, 30]
swap(p)
print (p)
```

```
[30,100]
```

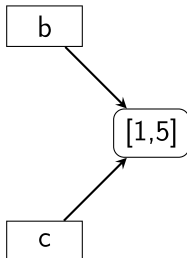
- Nell'esempio, le strutture dati complesse sono passate per riferimento ...
- ...se modifico il contenuto del parametro formale *g*, modifico la variabile originale *p*

Variabili e riferimento: generalizzazione

- In generale, le strutture dati complesse non sono altro che dei riferimenti a *porzioni di memoria*
- Esempio

```
b=[1,5]
c=b
print(c) # [1,5]
c[1]=10
print(b) # Cosa viene stampato?
         # [1,10]
```

- L'istruzione di riga 4 modifica l'area di memoria a cui sia *c* sia *b* fanno riferimento



Name Space, scoping delle variabili

Variabili locali, variabili globali

Variabili locali

- Variabili locali: variabili create all'interno di una funzione

```
1 def stampaUnione(parte1, parte2):
2       messaggio = parte1 + parte2
3       print(messaggio)
4 frase1 = "Nel mezzo "
5 frase2 = "del cammin"
6 stampaUnione(frase1, frase2)
```

```
Nel mezzo del cammin
```

- *messaggio* è una variabile locale
- *parte1* e *parte2* sono parametri formali
- I parametri formali sono assimilabili a delle variabili locali

Variabili locali 2

- Le variabili locali esistono solo all'interno della funzione
- Non possono essere usate all'esterno.

```
1 def stampaUnione(parte1, parte2):
2       messaggio = parte1 + parte2
3       print(messaggio)
4 frase1 = "Nel mezzo "
5 frase2 = "del cammin"
6 stampaUnione(frase1, frase2)
7 print(messaggio)
```

- Tutto a posto in questo codice?
- No! L'ultima istruzione, causa un messaggio d'errore

  NameError: messaggio

- Al termine della chiamata alla funzione *stampaUnione*, la variabile locale *messaggio* viene distrutta

Variabili globali

- Introduciamo le *variabili globali*, si tratta di variabili dichiarate nel corpo principale della funzione
  - Visibili all'interno di tutte le funzioni (dichiarate nello script)
  - Da qui il nome *variabili globali*
- Es.

```
1 PI=3.14  # variabile globale
2 def area(r):
3     return PI * r**2 # l'elevamento a potenza
4                      # ha la precedenza
5 l=2
6 a=area(l)
7 print("Un cerchio di raggio %f ha area %f" % (l, a) )
```

Scope delle variabili

- Scope (visibilità) di una variabile: l'insieme delle righe di codice in cui una variabile e il suo contenuto sono accessibili
- Namespace: l'insieme delle variabili alle quali l'interprete Python può accedere in un certo istante
- Il name space è dinamico: cambia durante l'esecuzione di uno script

```
a=10
def somma(p1, p2):
    risultato = p1+p2
    return risultato
b=20
c=somma(a,b)
print(c)
```

Esempi di Namespace

a=10

a = 10
b = 20
p1 = 10
p2 = 20
risultato = 30

a = 10
b = 20
c = 30

Name Space di una funzione

Ne fanno parte:

- le variabili definite all'interno della funzione:
  - variabili locali
  - parametri formali
- le variabili definite nel blocco di codice all'interno del quale è dichiarata la funzione (spesso è il corpo principale del programma)
  - variabili globali (le variabili definite nel corpo principale del programma)
  - ...

Esempio

```
1 def interessi(capitale, tasso):
2     i=capitale*tasso/100.0
3     return i
4 prestito = 20
5 t=5
6 c = interessi(prestito, t)
7 print(c)
```

Namespace

prestito = 20
t = 5
capitale = 20
tasso = 5
i = 1

Risoluzione dei conflitti

- E' possibile che si dichiari una variabile locale con un nome già utilizzato da una variabile globale ...
- ... ciò genera un *potenziale* conflitto
- Esempio

```
a=100    # variabile globale
def calcola(x):
    a=5 # variabile locale
    return 2*x
b=calcola(1)+a
print(b) # Cosa viene visualizzato?
```

- Secondo voi che cosa viene visualizzato a riga 6?

102

Spiegazione del comportamento

- Se all'interno di una funzione si definisce una variabile con lo stesso nome di una variabile globale (es. la variabile **a** dell'esempio precedente), la variabile all'interno della funzione maschererà la variabile globale, per tutta la durata dell'esecuzione della funzione
- (in altre parole) non potete modificare all'interno di una funzione il valore di una variabile globale
- Questo comportamento è voluto, serve per evitare di modificare inavvertitamente variabili globali
    - Si tratta di errori che in altri linguaggi possono accadere e ...
    - ...sono molto difficili da scovare

In una funzione . . .

- . . . è un buono stile di programmazione:
  - limitare l'accesso alle variabili globali
  - utilizzare solo i parametri formali per veicolare informazioni dall'esterno

1) Cattivo stile

```
1 prestito = 20
2 t=5
3 def interessi(capitale):
4     i=capitale*t/100.0
5     return i
6 c = interessi(prestito)
7 print(c)
```

2) Buono stile

```
1 prestito = 20
2 t=5
3 def interessi(capitale,tasso):
4     i=capitale*tasso/100.0
5     return i
6 c = interessi(prestito, t)
7 print(c)
```

- In 1) la modifica del nome della variabile globale *t* creerebbe problemi alla funzione *interessi*

Parametri opzionali

- Nei parametri formali di una funzione possono essere inseriti anche dei parametri opzionali

```
1 def interessi(capitale, tasso=0.05 ):
2     i=capitale*tasso
3     return i
4 c = interessi(prestito)
```

valore di default del parametro opzionale

- Esempio di utilizzo

```
1 s=interessi(100) # viene utilizzato il valore di default
2                  # se non e' assegnato un valore al param. opz.
3 print(s) # 5
4 s=interessi(100, tasso=0.12) # Nell'invocazione di
5                              # una funzione, i parametri opzionali
6                              # vanno sempre dopo i parametri obbligatori
7 print(s) # 12
```

- Nella slide precedente, *tasso* era stato dichiarato come parametro obbligatorio

Parametri opzionali 2

- Data una funzione definita come segue:

```
1 def fun1(par1, par2, opz1=100, opz2=False):
2     ...
```

- Come vengono accoppiati i parametri attuali con i parametri formali nell'invocazione seguente?

```
3 a=fun1(10, 20, opz2=True, opz1=2)
```

- I parametri obbligatori vengono mappati sulla base della posizione
  - 10 → par1
  - 20 → par2
- I parametri opzionali vengono mappati in base al nome del parametro utilizzato, indipendentemente dalla loro posizione
  - 2 → opz1
  - True → opz2

Namespace all'interno della funzione (dopo l'invocazione da riga 3)

par1=10
par2=20
opz1=2
opz2=True

Funzioni senza return

- Funzioni senza return restituiscono il valore speciale None
  - *None* è una costante speciale del linguaggio
  - *None* è logicamente equivalente a False
- Quindi, tutte le funzioni in Python hanno sempre un return value,
  - o il valore restituito dal *return* esplicitamente codificato dal programmatore
  - o il *None* che l'interprete python restituisce, quando la funzione termina senza aver incontrato un return esplicito.
  - In altre parole, è come se al termine di ogni funzione ci sia un *return None* che viene eseguito solo se l'interprete non incontra altri return durante l'esecuzione della funzione.
  - per questo motivo, quelle che in altri linguaggi sarebbero considerate *procedure*, in Python sono comunque classificate come funzioni

Return con più valori

- Cosa notate di strano nello script qua sotto?

```
def fun1(a,b):
    a+=1
    b+=1
    return a,b

c,d = fun1(5,10)
print(c) #  6
print(d) # 11
```

- Una funzione in python può restituire più valori

- Investighiamo il valore restituito

```
f=fun1(5,10)
print(f)
```

```
(6, 11)
```

- Vi ricorda qualcosa?

```
print(type(f))
```

```
<type 'tuple'>
```

- Quando vengono restitutiti più valori con l'istruzione return, python costruisce una tupla

- Gli script qua sotto sono equivalenti

```
1  def fun1(a,b):
2     a+=1
3     b+=1
4     return a,b
5
6  c,d = fun1(5,10)
7  print(c) #  6
8  print(d) # 11
```

```
10 def fun1(a,b):
11    a+=1
12    b+=1
13    return (a,b)
14
15 (c,d) = fun1(5,10)
16 print(c) #  6
17 print(d) # 11
```

```
20 def fun1(a,b):
21    a+=1
22    b+=1
23    return (a,b)
24
25 t = fun1(5,10)
26 c = t[0]
27 d = t[1]
28 print(c) #  6
29 print(d) # 11
```

- Reminder: in python, due o più valori separati da virgola costituiscono una tupla, anche senza parentesi tonde
- Data una tupla `t=(6,11)`
- L'istruzione `(c,d) = t` è equivalente a

  ```
  c = t[0]
  d = t[1]
  ```

Funzioni e comunicazione

Riepilogo di come una funzione comunica con il programma chiamante

- Dati in ingresso (dati che una funzione deve ricevere per svolgere l'elaborazione), accessibili tramite:
  - Parametri attuali
  - Variabili globali (sconsigliato)
- Dati in uscita. La funzione può inviare dati al programma chiamante usando
  - l'istruzione *return*
  - modificando variabili globali (sconsigliato). Attenzione: solo le strutture dati passate per riferimento mantengono le modifiche, dopo l'uscita dalla funzione (vedi esempio slide successiva)

Esercizio riepilogativo

```
1  def statistics(li, minmax):
2      min=li[0]
3      max=li[0]
4      sum=0
5      i=0
6      while i<len(li):
7          if li[i]<min:
8              min=li[i]
9          if li[i]>max:
10             max=li[i]
11         sum+=li[i]
12         i+=1
13     avg=sum/float(len(li))
14     minmax[0]=min
15     minmax[1]=max
16     return avg
17 values=[5,1,6,9,0]
18 m2=[None, None]
19 average=statistics(values, m2)
20 print(average) # 4.2
21 print(m2)      # [0, 9]
```

- Il parametro attuale *m2* è passato per riferimento
- Le modifiche fatte al parametro formale *minmax* si ripercuotono sul parametro attuale *m2*
- Ciò è stato usato per trasferire dei risultati dalla funzione al programma chiamante
- Per quanto possibile, quest'ultimo escamotage andrebbe evitato

- Perché l'uso dei parametri formali per restituire valori va evitato?
- Se non si sta molto attenti possono nascere dei problemi difficili da individuare
- La sostituzione delle parti commentate con l'istruzione a riga 16, causa il malfunzionamento del programma, vedi output della print()
- Qual è il motivo, secondo voi?

```
def statistics(li, minmax):
    min=li[0]
    max=li[0]
    sum=0
    i=0
    while i<len(li):
        if li[i]<min:
            min=li[i]
        if li[i]>max:
            max=li[i]
        sum+=li[i]
        i+=1
    avg=sum/float(len(li))
    # minmax[0]=min
    # minmax[1]=max
    minmax=[min,max] # Parte differ.
    return avg
values=[5,1,6,9,0]
m2=[None, None]
average=statistics(values, m2)
print(average) # 4.2
print(m2)      # [None, None]
```

- Con l'istruzione $minmax = [min, max]$, di fatto l'interprete python:
  - crea una nuova area di memoria in cui vengono copiati i valori delle due variabili *min* e *max* ...
  - ...e assegna a *minmax* l'indirizzo della nuova area di memoria.
  - Dato che i valori sono stati scritti su una nuova area di memoria, la (vecchia) area di memoria a cui fa riferimento $m2$ rimane inalterata (continua ad esserci $[None, None]$).
- Questo meccanismo è un'altro esempio di come Python cerca di proteggere le variabili globali (in questo caso $m2$) da modifiche fatte inavvertitamente dentro le funzioni.
- Il problema in questo caso è che il meccanismo di cui sopra fa assumere al programma un comportamento (apparentemente) controintuitivo.
- Nella prima versioen del programma (dove non c'è $minmax = [...]$ ma $minmax[0] = min$, allora *minmax* va ad operare sull'area di memoria a cui fa riferimento m2
- Osservazione: se per restituire i valori si fosse fatto uso del return, il problema non si sarebbe posto!