Python

In questa lezione:

- Le funzioni
- Parametri in ingresso
- Valori in uscita
- Variabili locali e globali
- Parametri per valore e per riferimento
- Funzioni ricorsive

Funzioni

Motivazione

Un modo per affrontare un problema difficile è quello di scomporlo in sottoproblemi e risolvere ciascuno di essi separatamente (divide et impera). Così nei programmi, visti come soluzioni a problemi complessi, è spesso utile avere sottoprogrammi che risolvono sottoproblemi.
I sottoproblemi possono essere a loro volta scomposti in altri più semplici in un procedimento di semplificazione che può essere reiterato più volte.
Questa strategia permette di:

- non perdere di vista la strategia globale
- concentrarsi su un sottoproblema indipendentemente dagli altri

In Python i sottoprogrammi corrispondono alle funzioni e sono sequenze di istruzioni dotate di un nome.

Funzioni

Invocazione

Abbiamo già invocato funzioni per risolvere dei sottoproblemi.
Per esempio:

- `print`: per stampare dei valori
- `len`: per ottenere la lunghezza di una stringa o di una lista
- `sqrt`: per ottenere la radice quadrata di un numero

Per invocazione di una funzione si intende il suo richiamo (tramite il nome) a cui corrisponde l'esecuzione delle istruzioni della funzione stessa.

Alcune di queste funzioni fanno parte del linguaggio, altre possono essere utilizzate solo dopo aver dichiarato l'uso delle rispettive librerie .

Se un sottoproblema è stato già affrontato da altri programmatori ed esiste la corrispondente funzione per risolverlo non resta che invocare la funzione!

Funzioni

Definizione

Quando un sottoproblema non ha ancora una soluzione è il momento di definire una funzione che lo risolve.
Poniamoci il semplice problema di voler stampare due righe contenenti il solo carattere "-" ripetuto 20 volte.
Possiamo definire una funzione `righe` che svolge questo compito. In Python si definisce una funzione utilizzando la parola chiave `def` e specificando il corpo, cioè le istruzioni che la compongono.

```
#definizione della funzione

def righe() :              # intestazione
      print("-" * 20)      # inizio del corpo della funzione
      print("-" * 20)      #


#programma
righe()            # invocazione della funzione (stampa due righe)
print("ciao")      # stampa ciao
righe()            # stampa altre due righe
```

Parametri in ingresso

Motivazioni

Molte funzioni operano in modo parametrico, cioè sulla base di valori che vengono forniti in ingresso alla funzione. Questi valori vengono detti argomenti della funzione.

Per esempio, per calcolare la radice quadrata di 2 posso invocare la funzione `sqrt` in questo modo:

```
x = sqrt(2)
```

In questo esempio 2 è l'argomento fornito alla funzione `sqrt`.

Gli argomenti, cioè i valori forniti alla funzione nel momento dell'invocazione, vengono anche detti parametri attuali.

Parametri in ingresso

Per memorizzare i valori degli argomenti prima della esecuzione delle istruzioni di una funzione si utilizzano delle variabili dette parametri della funzione (o parametri formali o anche variabili parametro). Queste variabili sono utilizzate dalle istruzioni nel corpo delle funzioni.

Quindi per definire una funzione dobbiamo scegliere il nome, ma anche definire una variabile per ciascuno dei suoi argomenti.
Supponiamo di voler modificare la funzione `righe` in modo tale che il carattere ripetuto 20 volte sia una argomento. Ecco la nuova definizione:

```
def righe(car) :       # car è la variabile parametro
      print(car * 20) # stampa car 20 volte
      print(car * 20) # idem


#programma
righe("#")        # stampa due righe con "#"
print("ciao")     # stampa ciao
righe("+")        # stampa due righe con "+"
```

Parametri in ingresso

Valori predefiniti

E' possibile definire dei parametri formali con valori predefiniti. In questo caso l'argomento è facoltativo e se non viene specificato assume il valore predefinito.
Es.:

```
def righe(car = "-", num_car = 20) : # parametri con valori predefiniti
      print(car * num_car)
      print(car * num_car)


#programma
righe()              # stampa due righe con 20 "-"
righe("#")           # stampa due righe con 20 "#"
righe("*", 30)       # stampa due righe con 30 "*"
righe(num_car = 10) # stampa due righe con 10 "-"
righe(num_car = 10, car = "-")  # come sopra
```


Valori in uscita

I sottopogrammi, oltre che per compiere delle azioni, sono spesso utilizzati per calcolare dei valori. Una funzione può terminare restituendo un valore. A questo scopo in Python si usa l'enunciato `return`.
L'enunciato `return` può essere utilizzato solo all'interno del corpo di una funzione e ha l'effetto di bloccare immediatamente l'esecuzione della funzione e di restituire il valore dell'espressione che segue.
Esempio:

```
def cubo( x ) :
      return x ** 3     # restituzione del cubo di x

def media( lista ) :
      somma_elementi = sum( lista )
      numero_elementi = len( lista )
      media = somma_elementi / numero_elementi
      return media      # restituzione della media dei valori in lista

l = [ ]
for i in range(1000):
   l.append( cubo(i) )

print( media(l) )  # media dei cubi dei numeri tra 0 e 999
```

Valori in uscita

In Python è possibile restituire più di un valore in uscita utilizzando le tuple.

Creiamo una funzione che tasforma coordinate polari in coordinate cartesiane:

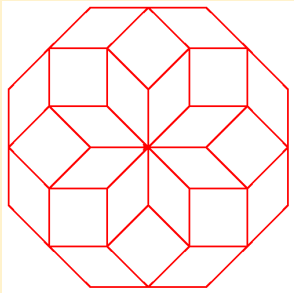
```
from math import sin, cos, pi

def cartesiane( distanza, angolo ) :
        ascissa  = distanza * cos(angolo)
        ordinata = distanza * sin(angolo)

        return ascissa, ordinata # restituisce una tupla di due elementi

r = 10       # r ed alfa solo le coordinate polari di un punto
alfa = pi/3

x, y = cartesiane(r, alfa)  # x ed y solo le coordinate cartesiane

print( x, y )
```


Funzioni

Altri esempi

Vediamo un esempio di procedura in turtleGraphic:

```
from turtle import Turtle

LATO = 100

def poligono(lati):        # disegna un poligono regolare di n lati
  for i in range(0,lati) :
     tarta.forward(LATO)
     tarta.left(360 / lati)


n = int(input("lati: "))  # per n = 8 :

tarta = Turtle()
tarta.width(3)
tarta.color("red")

for i in range(0, n):   # disegna un fiore con n poligoni di n lati
    poligono(n)
    tarta.left(360 / n)
```



Funzioni

Altri esempi

Vediamo un esempio di funzione booleana:

```
def contiene7( numero ):    # ritorna True se numero contiene la cifra 7
    while numero > 0 :
      if numero % 10 == 7:
          return True
      else:
          numero = numero // 10

    return False


numeri_con_7 = []

for i in range(0, 1000): # crea lista di numeri tra 0 e 999 contenenti 7
    if contiene7(i) :
      numeri_con_7 = numeri_con_7 + [i]

for el in numeri_con_7:  # stampa gli elementi della lista
    print(el)
```


Procedure e Funzioni

In alcuni linguaggi di programmazione (es. il Pascal), si fa differenza fra sottopogrammi che ritornano un valore e quelli che non ritornano nessun valore. Questi ultimi vengono chiamati procedure mentre i primi funzioni.

Ad esempio, `poligono` sarebbe una procedura, mentre `contiene_7` una funzione.

In Python, tutti i sottoprogrammi sono funzioni. Infatti, anche quando un sottoprogramma non termina con l'enunciato `return`, comunque viene restituito un valore. Questo valore è `None`, che vuol dire "niente", "nessun valore".

`None` può anche essere assegnato ad una variabile per dire che questa, al momento, non si riferisce a nulla: è il valore per un riferimento nullo.

Variabili locali

Le variabili definite all'interno di una funzione si chiamano variabili locali. Quando una variabile è definita all'interno di un corpo di una funzione rimane visibile solo fino alla fine della funzione.

La visibilità di una variabile locale non si estende al di fuori della funzione. Le variabili parametro (cioè i parametri formali) sono variabili locali della funzione.

```
x = 10
y = 9

def fun():
     y = 8            # y è una nuova variabile locale
     z = 7            # variabile locale
     print(x, y, z)   # stampa 10 8 7 (x è visibile!)


fun()              # stampa 10 8 7
print(x, y)        # stampa 10 9
print(x, y, z)     # Errore! z non è visibile
```


Variabili globali

Le variabili definite al di fuori di una funzione si chiamano variabili globali. Queste sono visibili a tutte le funzioni definite successivamente. È il caso della variabile `x` dell'esempio precedente.

Ma se all'interno di una funzione si tenta di assegnare un valore ad una variabile globale, in realtà si crea una nuova variabile locale con lo stesso nome della globale. È il caso della variabile `y` dell'esempio precedente. Come si può modificare il valore di una variabile globale? In Python si deve dichiararlo con l'enunciato `global`.

```
x = 10  # variabile globale
y = 9   # variabile globale

def fun():
     global y     #
     y = 8        # y viene modificato!

print(x, y)       # stampa 10 9
fun()
print(x, y)       # stampa 10 8
```

Parametri per valore e per riferimento

Passaggio di valori

L'invocazione di una funzione con parametri prevede:

- valutazione delle espressioni che determino il valore degli argomenti
- assegnazione dei valori ottenuti ai rispettivi parametri formali
- esecuzione delle istruzioni del corpo della funzione

```
a = 10
b = 10

def cubo( x ) :
      return x ** 3

c = cubo(2)       # espressione: 2     valore asseganto ad x: 2
d = cubo(a)       # espressione: a     valore asseganto ad x: 10
e = cubo(a + b)   # espressione: a + b valore asseganto ad x: 20
```

A meno di non usare `global` una funzione non può cambiare il valore di variabili esterne, perché opera solo su variabili locali e variabili parametro che sono comunque variabili locali.

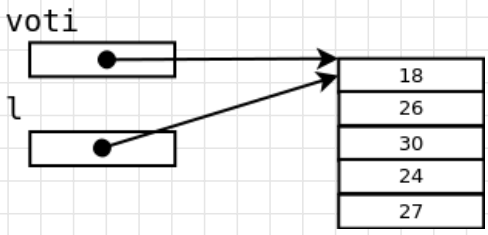
Ma cosa succede quando il valore dell'argomento è un riferimento?

Parametri per valore e per riferimento

Passaggio di riferimenti

Nel caso di invocazione di una funzione con un argomento che sia un riferimento, il valore assegnato al parametro formale è il valore del riferimento:

```
voti = [18, 26, 30, 24, 27]

def magari( l ) :
      l[0] = 30

print(voti)  # [18, 26, 30, 24, 27]
magari(voti) # cambia l[0] !!!
print(voti)  # [30, 26, 30, 24, 27]
```



Il valore di `voti`, cioè il riferimento alla lista non può essere cambiato all'interno della funzione, ma tramite questo riferimento si può cambiare il valore degli elementi della lista!

Funzioni ricorsive

Un problema può essere risolto da una funzione che svolge una parte del compito e poi richiama se stessa per la restante parte.

Una funzione ricorsiva è una funzione che invoca se stessa.

Per evitare che una funzione richiami se stessa all'infinito, è necessario che termini almeno in caso, detto caso base, mentre per gli altri casi può richiamare se stessa nel passo ricorsivo.

Molte concetti matematici possono essere espressi in modo ricorsivo. Tra questi:

- il fattoriale di un numero
- numeri di Fibonacci
- le operazioni di addizione e moltiplicazione
- le serie matematiche

Il fattoriale

Il fattoriale di un numero naturale $n > 0$ (denotato con $n!$) è il prodotto dei naturali da 1 fino ad n. Per definizione $0! = 1$.

$$n! = \begin{cases} 1 & n = 0 \\ \prod_{i=1}^{n} i & n > 0 \end{cases}$$

Il fattoriale si presta ad una definizione ricorsiva:

$$n! = \begin{cases} 1 & n = 0 \\ n \cdot (n-1)! & n > 0 \end{cases}$$

a cui immediatamente corrisponde la seguente funzione Python:

```
def fatt(n) :
    if n == 0 :
        return 1              # caso base
    else:
        return n * fatt(n-1) # passo ricorsivo
```

Numeri di Fibonacci

I numeri di Fibonacci sono i numeri della successione:
$1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$
L'ennesimo numero di Fibonacci F_n è la somma dei due precedenti e, per definizione, $F_0 = 1$ e $F_1 = 1$.

$$F_n = \begin{cases} 1 & n = 0 \\ 1 & n = 1 \\ F_{n-1} + F_{n-2} & n > 1 \end{cases}$$

```
def F(n) :
    if n == 0 or n ==1 :
        return 1                # caso base
    else:
        return  F(n-1) + F(n-2) # passo ricorsivo
```


Ricorsione: un altro esempio

Si possono svolgere anche delle operazioni pensando in modo ricorsivo.
Supponiamo di voler sommare gli elementi di una lista di numeri reali.
Si può dire che la somma è 0 se la lista è vuota, altrimenti è il primo numero della lista più la somma dei restanti.

$$somma(\mathrm{L}) = \begin{cases} 0 & \mathrm{L} = [\,] \\ \mathrm{L}[0] + somma(\mathrm{L}[1:]) & \mathrm{L} \neq [\,] \end{cases}$$

Ecco una definizione in Python:

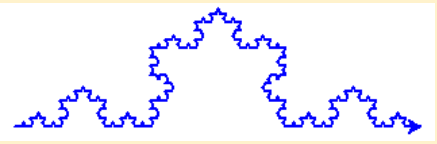
```
def somma(l) :
    if l == [] :
        return 0                  # caso base
    else:
        return  l[0]+somma(l[1:]) # passo ricorsivo
```

Ricorsione con turtleGraphics

Anche alcune curve sono definite in modo ricorsivo.
Una di queste ricorsivamente sostituisce un segmento

 con  .

```
from turtle import Turtle

MinDist = 10

t = Turtle(); t.color("blue"); t.width(2)

def lato(l):
    if l < MinDist :
        t.forward(l)  # caso base
    else:
        lato(l/3)     # passo
        t.left(60)    # ricorsivo
        lato(l/3)
        t.right(120)
        lato(l/3)
        t.left(60)
        lato(l/3)

lato(300)
```



Funzioni

Esercizi sulle funzioni

Utilizzando le funzioni risolvere i seguenti esercizi:

- Realizzare una funzione che ricevendo in ingresso un naturale e una base lo scriva in quella base.
- Realizzare una funzione che ricevendo in ingresso due coordinate cartesiane le trasformi in polari.
- Due numeri sono coprimi se non hanno numeri in comune. Realizzare una funzione per controllare se due numeri sono coprimi.
- Realizzare una funzione che ricevendo due liste in ingresso controlli se abbiano o meno un elemento in comune.
- Realizzare una funzione che ricevendo due matrici in ingresso controlli se siano una la trasposta dell'altra.
- Realizzare una funzione ricorsiva che calcoli la somma dei primi n naturali.
- (turtleGraphic): Realizzare una funzione ricorsiva che nel passo base tracci un segmento AB e lo sostituisca ricorsivamente con due segmenti AC, CB dove C è il centro del quadrato di alto AB.