Complementi di Programmazione

Python: Funzioni

CdL Informatica - Università degli studi di Modena e Reggio Emilia
AA 2023/2024

Filippo Muzzini

Funzioni - definizione

In Python esistono le funzioni. A differenza dei metodi esse non sono innestate in una classe.

Si definiscono con la parola chiave **def**

La funzione è un blocco di codice (quindi indentato)

```
def funzione(arg1, arg2):
    print(arg1)
    print(arg2)
    return 1
```

Funzioni - definizione

In Python esistono le funzioni. A differenza dei metodi esse non sono innestate in una classe.

Si definiscono con la parola chiave **def**

La funzione è un blocco di codice (quindi indentato)

```
def funzione(arg1, arg2):
    print(arg1)
    print(arg2)
    return 1
```

Non si specificano i tipi degli argomenti
e dei valori ritornati dalla funzione!

Funzioni - esempio

```
def fib(n):
    a, b = 0, 1
    print(a)
    for i in range(n):
        print(b)
        a, b = b, a + b
```

Funzioni - Argomenti opzionali

E’ possibile definire funzioni con parametri opzionali che assumono un valore di default se non passati durante l’invocazione.

```
def funzione(arg1, arg2=0):
    print(arg1)
    print(arg2)
    return 1
```

Si specifica il valore di default usando **=**
I parametri opzionali devono essere gli ultimi.

Funzioni - Passaggio di argomenti

Normalmente si usa la notazione posizionale (positional arguments)


```
    print(arg1)
    print(arg2)
    return 1
```

Funzioni - Passaggio di argomenti

E’ possibile anche specificare a quale parametro ci si riferisce (keyword arguments)


```
def funzione(arg1, arg2=0):
    print(arg1)
    print(arg2)
    return 1
```

Funzioni - Passaggio di argomenti

E’ possibile mischiare entrambe le tecniche.

I positional argument vanno prima dei keyword argument in questo caso!

```
a = funzione(3,arg2=1)

def funzione(arg1, arg2=0):
        print(arg1)
        print(arg2)
        return 1
```

Funzioni - Passaggio di argomenti

E’ possibile ‘spacchettare’ sequenze (es. liste) e passare ogni elemento come positional argument.

Si usa l’**operatore * anteposto al dizionario**

a = [1,2,3,4,5]

funzione(*a) equivale a **funzione(1,2,3,4,5)**

Il numero di parametri deve essere lo stesso di quelli definiti nella funzione (al netto di quelli opzionali)

Funzioni - Passaggio di argomenti

E’ possibile ‘spacchettare’ dizionari e passare ogni elemento come keyword argument.

Si usa l’**operatore ** anteposto alla dizionario**

a = {a:1, b:2, c:3}

funzione(a)** equivale a **funzione(a=1,b=2,c=3)**

Funzioni - Parametri formali con * e **

Gli operatori * e ** possono anche essere utilizzati nella definizione della funzione.

```
def funzione(*args, **kwargs):
    …
```

Funzioni - Parametri formali con * e **

Gli operatori * e ** possono anche essere utilizzati nella definizione della funzione.

def funzione(*args, **kwargs):

    …

In questo caso tutti i parametri posizionali passati alla funzione vengono condensati nella tupla **args**.

Tutti i keyword arguments vengono condensati in un dizionario **kwargs**

Funzioni - Parametri formali con * e **

Gli operatori * e ** possono anche essere utilizzati nella definizione della funzione.

def funzione(*args, **kwargs):

    …

Utile se non si sa a priori il numero di parametri necessari.

Funzioni - Parametri formali con * e **

Gli operatori * e ** possono anche essere utilizzati nella definizione della funzione. **La funzione può comunque avere parametri espliciti**

def funzione(arg1, arg2, *args, **kwargs):

    …

arg1 e arg2 non saranno presenti in args e kwargs

Funzioni - Ricorsione

Come in altri linguaggi, in Python è possibile scrivere una funzione che chiama se stessa. **Ricorsione**

Funzioni - Ricorsione

Come in altri linguaggi, in Python è possibile scrivere una funzione che chiama se stessa. **Ricorsione**

Remainder:

- CONDIZIONE DI STOP!

Funzioni - Ricorsione

```
def fib(n):
    a, b = 0, 1
    print(a)
    for i in range(n):
        print(b)
        a, b = b, a + b
```

```
def fib(n):
    if (n == 0):
        print(0)
        return 0,1
    a, b = fib(n-1)
    print(a+b)
```

Funzioni - Ricorsione - Appiattire le liste

Provare usando la ricorsione a scrivere una funzione che appiattisca una lista di liste.

Aiutarsi con la funzione **hasattr.** Cercare nella documentazione

Funzioni - Ricorsione - Appiattire le liste

Provare usando la ricorsione a scrivere una funzione che appiattisca una lista di liste.

```
def flatten(l):
    res = []
    if not hasattr(l, '__iter__'):
        return [l]
    for o in l:
        res.extend(flatten(o))
    return res
```

Programmazione di ordine superiore

Si definisce un linguaggio come **di ordine superiore** se consente di utilizzare funzioni come valori.

Le funzioni possono essere passate ad altre funzioni

Possono essere ritornate come risultato di altre funzioni

Programmazione di ordine superiore

```
def get_print_help_function(lang):
    def eng_help():
        print(‘help’)
    def ita_help():
        print(‘aiuto’)
    if lang == ‘eng’:
        return eng_help
    else:
        return ita_help
```

Il vaolore di ritorno è una funzione.
Notare l’assenza di **()**

Programmazione di ordine superiore

```
def print_function_result(func):
    print(f'il risultato è {func()}')
```

La funzione prende in ingresso una funzione che chiamerà al suo interno.

Programmazione di ordine superiore - CLOSURE

Supponiamo di voler avere una funzione inizializzabile che poi stampi sempre lo stesso messaggio quando chiamata.

Si potrebbe creare una funzione che preso il parametro di inizializzazione restituisce una funzione che stampa sempre quel messaggio.

Programmazione di ordine superiore - CLOSURE

Si potrebbe creare una funzione che preso il parametro di inizializzazione restituisce una funzione che stampa sempre quel messaggio.

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer
```

Programmazione di ordine superiore - CLOSURE

Si potrebbe creare una funzione che preso il parametro di inizializzazione restituisce una funzione che stampa sempre quel messaggio.

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer
```

Printer deve accedere a **msg** che non fa parte della funzione stessa.

In Python è possibile

Programmazione di ordine superiore - CLOSURE

Si potrebbe creare una funzione che preso il parametro di inizializzazione restituisce una funzione che stampa sempre quel messaggio.

```
def print_msg(msg):
    def printer():
        print(msg)
    return printer
```

Concetto di **CLOSURE:**
Le funzioni innestate possono accedere alle variabili delle funzioni madri.
Tali variabili vengono incapsulate nelle funzioni figlie in modo che possano essere utilizzate anche quando la funzione madre termina.

Programmazione di ordine superiore - Lambda

Partiamo da una funzione di Python molto esplicativa del concetto di ordine superiore.

La funzione **map()**

Essa prende in ingresso una funzione e una sequenza di oggetti.

Applica tale funzione e ogni oggetto e ne ritorna il risultato (come lista)

Programmazione di ordine superiore - Lambda

In certi casi la funzione da applicare è molto semplice, per esempio il quadrato.

```
def quad(a)
    return a**2


map(quad, [1,2,3])
```

Programmazione di ordine superiore - Lambda

Python permette di passare la funzione senza definirla in un altro punto del codice.

Viene detta lambda function (o anche funzione anonima)

Sintassi: **lambda** argomenti: valore di ritorno

Può contenere un'unica istruzione che sarà il valore di ritorno (non si mette return)

map(**lambda x:x**2**, [1,2,3])

Generatori

Abbiamo visto che in Python si può iterare su qualsiasi oggetto che abbia un metodo __iter__ che ritorni un iteratore.

Questo implica la costruzione di una classe con tale metodo

Generatori

I **Generatori** semplificano questo lavoro.

Possono essere sia funzioni sia espressioni

In entrambi i casi possono essere utilizzate in un ciclo

Generatori - Funzioni generatrici

Le funzioni Generatrici generato ad ogni iterazione l’elemento successivo.

E come se ritornassero una serie di valori invece che un unico risultato.

Si creano attraverso il comando **yield seguito dal valore da tornare**

Generatori - Funzioni generatrici

Quando una funzione contiene il comando yield, essa viene trattata da Python come un Generatore.

Implicitamente viene dotata dei metodo __iter__ e __next__

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```


vien chiamata la funzione

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

inizia l’esecuzione

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

viene ritornato il primo valore e si interrompe l’esecuzione della funzione

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```


y assume tale valore

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```


viene utilizzato il valore e termina
la prima iterazione del ciclo

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```


inizia la seconda iterazione

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

viene ripresa l'esecuzione della funzione da dove si era interrotta

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

viene ripresa l'esecuzione della funzione da dove si era interrotta

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

viene ritornato il secondo valore

Generatori - Funzioni generatrici

Esempio: range() fai da te!

```
def range(stop):
    i = 0
    while i < stop:
        yield i
        i = i + 1
```

```
for y in range(10):
    …
```

Alla fine di tutto viene sollevata l'eccezione StopIteration per fermare il ciclo

Generatori - Espressioni generatrici

I Generatori possono anche essere creati con una sintassi molto stringata.

Si parla di **espressioni generatrici**

Sintassi: **(**espressione **for** variabili **in** sequenza**)**

Esempio: **(x**2 for x in range(10))**

il generatore ritornerà ad ogni iterazione il quadrato dei numeri da 0 a 9

List comprehension

Una sintassi simile (usando [] invece che ()) può essere usata per generare liste in modo immediato:

a = [x2 for x in range(10)]**

a conterrà una lista dei quadrati da 0 a 9