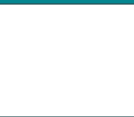




**University  
of Ferrara**

# Python

Funzioni Lambda, scope



# Funzioni Lambda

- Quando si definisce una funzione con la clausola `def`, le si assegna automaticamente una variabile.
- In Python, le funzioni possono essere utilizzate anche **on the fly** (create e utilizzate allo stesso tempo come un `int` o una stringa), utilizzando una particolare sintassi chiamata **lambda**.
  - Le funzioni che vengono create in questo modo vengono chiamate **anonime**.
- Questo approccio si utilizza spesso quando si vuole passare una funzione come argomento di un'altra funzione.
- La sintassi `lambda` richiede la clausola `lambda` seguita da un elenco di argomenti, un carattere di due punti, e l'espressione per valutare gli argomenti.

**lambda x : x**

# Funzioni Lambda

- Quando si definisce una funzione con la clausola `def`, le si assegna automaticamente una variabile.
- In Python, le funzioni possono essere utilizzate anche **on fly** (create e utilizzate allo stesso tempo come un `int` o una stringa), utilizzando una particolare sintassi chiamata **lambda**.
  - Le funzioni che vengono create in questo modo vengono chiamate **anonime**.
- Questo approccio si utilizza spesso quando si vuole passare una funzione come argomento di un'altra funzione.
- La sintassi lambda richiede la clausola `lambda` seguita da una lista di argomenti, un carattere di due punti, e l'espressione che restituisce il valore per i argomenti.

```
lambda x : x
```

Questa funzione lambda semplicemente restituisce il valore preso in ingresso.

# Funzioni Lambda

- Quando si definisce una funzione con la clausola `def`, le si assegna automaticamente una variabile.
- In Python, le funzioni possono essere create e utilizzate allo stesso modo, ma con una particolare sintassi (create e utilizzando una **anonima** funzione).
  - Le funzioni che vengono create in questo modo sono dette **funzioni anonime**.
- Questo approccio è utile quando si vuole creare una funzione come argomento di una funzione.
- La sintassi lambda è composta da un elenco di argomenti, un colon, un'espressione e un corpo della funzione.

## ATTENZIONE!

Queste funzioni possono prendere un numero qualsiasi di argomenti in ingresso, ma possono eseguire solo funzioni descrivibili con un'espressione singola (equivalente ad una linea di codice).

```
lambda x : x
```

# Funzioni Lambda

```
m1 = lambda x : x  
print(m1(1))
```

```
m1 = lambda x,y : x + y  
print(m1(1,2))
```

# Funzioni Lambda

```
m1 = lambda x : x  
print(m1(1))
```



Output: 1

```
m1 = lambda x,y : x + y  
print(m1(1,2))
```

# Funzioni Lambda

```
m1 = lambda x : x  
print(m1(1))
```

```
m1 = lambda x,y : x + y  
print(m1(1,2))
```



Output: 3

# Funzioni Lambda

- Il loro potere è dovuto al fatto che possiamo utilizzarle all'interno di altre funzioni per definire un template per le funzioni.

```
def myfunc(n):  
    return lambda a : a ** n
```

```
square = myfunc(2)  
print(square(3))
```

```
cube = myfunc(3)  
print(cube(3))
```



# Funzioni Lambda

- Il loro potere è dovuto al fatto che possiamo utilizzarle all'interno di altre funzioni per definire un template per le funzioni.

```
def myfunc(n):  
    return lambda a : a ** n
```

```
square = myfunc(2)  
print(square(3))
```

```
cube = myfunc(3)  
print(cube(3))
```



Output: 9

# Funzioni Lambda

- Il loro potere è dovuto al fatto che possiamo utilizzarle all'interno di altre funzioni per definire un template per le funzioni.

```
def myfunc(n):  
    return lambda a : a ** n
```

```
square = myfunc(2)  
print(square(3))
```

```
cube = myfunc(3)  
print(cube(3))
```



Output: 27

# Scope

- Per comprendere l'ambito di azione delle variabili (scope), è importante prima conoscere cosa sono realmente le variabili.
- Essenzialmente, esse sono riferimenti, o puntatori, ad un oggetto nella memoria.
- Quando si esegue un assegnamento di una variabile con = ad un'istanza, si lega (si fa il mapping) la variabile all'istanza.
- Più di una variabile può essere assegnata alla stessa istanza (aliasing).

# Scope

- Python tiene traccia di questi mappings con un namespace.
- Questi sono contenitori per il mapping tra i nomi delle variabili e gli oggetti associati.
- E' possibile pensare ad essi come dei dizionari, contenente la mappatura nome:oggetto.
- Nell'esempio che segue, i viene prima assegnato all'intero 5. In questo caso i è il nome della variabile, mentre l'intero di valore 5 è l'oggetto.
- Successivamente, j viene posta uguale ad i. Ciò significa che j è ora collegata allo stesso intero a cui è legato i, ovvero 5.

# Scope

- Poi si cambia i ponendolo uguale a 3, un programmatore inesperto potrebbe pensare che anche j ora sia uguale a 3, ma non è questo il caso.
- j è ancora legato (o sta ancora puntando) all'intero di valore 5. L'unica cosa che è cambiato è i, che ora è collegata all'intero di valore 3.

```
i = 5
```

```
j = i
```

```
i = 3
```

```
print("i: " + str(i))
```

```
print("j: " + str(j))
```

```
i: 3 j: 5
```

# Scope

- Se si definisce una variabile all'inizio dello script, essa sarà una variabile globale.
- Ciò significa che essa sarà accessibile ovunque all'interno dello script, quindi anche all'interno delle funzioni.

```
a = 5
```

```
def function():
```

```
    print(a)
```

```
function()
```

```
print(a)
```

```
5
```

```
5
```

# Scope

- Nel prossimo esempio, `a` è definito globalmente come 5, ma poi viene definito nuovamente come 3, all'interno della funzione.
- Se si stampa il valore di `a` dall'interno della funzione, verrà stampato il valore definito localmente all'interno della funzione.
- Se si stampa `a` al di fuori della funzione, verrà stampato il valore globale.
- La `a` definita in `function()` è letteralmente isolata dal mondo esterno.
- Vi si può accedere solo localmente, dall'interno della funzione stessa. Perciò le due `a` sono differenti tra loro, l'accesso cambia a seconda di dove avviene.

# Scope

```
a = 5
def function():
    a = 3
    print(a)
function()
print(a)
```

3

5



# Scope

- Supponiamo, quindi, che si abbia un'applicazione che ricorda un nome, il quale può anche essere modificato con una funzione `change_name()` .
- Il nome della variabile viene definito globalmente e localmente all'interno della funzione. Come si può notare, la funzione fallisce nel tentativo di modificare la variabile globale.

# Scope

```
name = 'Théo'
def change_name(new_name):
    name = new_name
print(name)
change_name('Karlijn')
print(name)
```

Théo

Théo

# La parola chiave global

- Con `global`, si indica a Python di usare le variabili globali invece delle variabili locali.
- Per usarlo, è sufficiente scrivere `global`, seguito dal nome della variabile.
- In questo caso, il nome di una variabile globale viene modificato usando la funzione `change_name()`.

# La parola chiave global

```
name = 'Théo'
def change_name(new_name):
    global name
    name = new_name
print(name)
change_name('Karlijn')
print(name)
```

Théo

Karlijn

# La parola chiave nonlocal

- L'istruzione nonlocal è utile per le funzioni innestate.
- Essa consente ad una variabile di riferirsi ad un'altra variabile assegnata all'interno dello scope più vicino.
- In altre parole, eviterà che la variabile tenti di legarsi prima localmente e la costringerà a passare a un livello "più alto"

# La parola chiave nonlocal

```
x = "a"
def outer():
    x = "b"
    def inner():
        x = "c"
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
```

```
inner: c
outer: b
global: a
```

# La parola chiave nonlocal

```
x = "a"
def outer():
    x = "b"
    def inner():
        nonlocal x
        x = "c"
        print("inner:", x)
    inner()
    print("outer:", x)
outer()
print("global:", x)
```

```
inner: c
outer: c
global: a
```

# Le closures in Python

- Le closures sono oggetti funzione che ricordano i valori negli scope in cui sono incluse, anche se non sono più presenti in memoria.
- Ricorda che una funzione nidificata è una funzione definita in un'altra funzione, come `inner()` è definita all'interno `outer()` nell'esempio seguente.
- Le funzioni nidificate possono accedere alle variabili dell'ambito di inclusione, ma non possono modificarle, a meno che non utilizzi `nonlocal`.



# Le closures in Python

```
def outer(number) :  
    def inner() :  
        number = 3  
        print("Inner: " + str(number) )  
    inner()  
    print("Outer: " + str(number) )  
outer(9)
```

Inner: 3

Outer: 9

# Le closures in Python

```
def outer(number):  
    def inner():  
        nonlocal number  
        number = 3  
        print("Inner: " + str(number))  
    inner()  
    print("Outer: " + str(number))  
outer(9)
```

Inner: 3

Outer: 3

# Le closures in Python

- Si vuole conservare una variabile definita in una funzione annidata, senza dover modificare una variabile globale.
- In Python, una funzione è anche considerata un oggetto, il che significa che può essere restituita e assegnata a una variabile.
- Nel prossimo esempio, si vedrà che invece di chiamare `inner()` all'interno di `outer()`, viene utilizzato `return inner`.
- Quindi, `outer()` viene chiamato con un argomento stringa e assegnato a un chiusura.
- Ora, anche se le funzioni `inner()` e `outer()` hanno terminato l'esecuzione, il loro messaggio è ancora conservato. Chiamando `closure()`, il messaggio può essere stampato.

# Le closures in Python

```
def outer(message):  
    # enclosing function  
    def inner():  
        # nested function  
        print(message)  
    return inner  
closure = outer("Hello world!")  
closure()
```

Hello world!

# Le closures in Python

- **Nota** che se chiami chiusura senza parentesi, verrà restituito solo il tipo dell'oggetto. Si vede che è del tipo

`closure`

`<function __main__.outer.<locals>.inner>`

# Le regole LEGB

- Come hai visto prima, gli spazi dei nomi possono esistere indipendentemente l'uno dall'altro e avere determinati livelli di gerarchia, che chiamiamo scope.
- A seconda di dove ti trovi in un programma, verrà utilizzato uno spazio dei nomi diverso.
- Per determinare in quale ordine Python accede agli spazi dei nomi, puoi utilizzare la regola LEGB.
- LEGB sta per:
  - Local
  - Enclosed
  - Global
  - Built-in

# Le regole LEGB

- Supponiamo che tu stia chiamando `print(x)` in `inner()`, che è una funzione nidificata in `outer()`. Quindi Python cercherà prima se `x` è stato definito localmente in quel `inner()`.
- In caso contrario, verrà utilizzata la variabile definita in `outer()` . Questa è la funzione enclosing.
- Se anche lì non è stato definito, l'interprete Python salirà di un altro livello, fino all'ambito globale.
- Oltre a questo troverai solo lo scope built-in, che contiene variabili speciali riservate per Python stesso.

# The LEGB rule

```
# Global scope
```

```
x = 0
```

```
def outer():
```

```
    # Enclosed scope
```

```
    x = 1
```

```
    def inner():
```

```
        # Local scope
```

```
        x = 2
```



# NumPy ed altri moduli utili

- sys
- NumPy
- Scikit-learn
- Managing CSVs (csv and Pandas)

# Modulo sys

- E' un modulo integrato formato da funzioni e parametri utili ad interagire con il sistema operativo.
- Solitamente viene utilizzato per ottenere gli argomenti dalla linea di comando

```
import sys
```

```
print("Script name: " + sys.argv[0])
```

```
for i in range(1, len(sys.argv)):
```

```
    print("Arg " + str(i) + ":" + sys.argv[i])
```




argv[0] nome del file

# Modulo sys

- E' possibile usarlo per ridirezionare il processo verso lo stdIO.

```
import sys
save_stdout = sys.stdout # keep previous streams
save_stderr = sys.stderr
file1 = open("output.log", "w")
file2 = open("error.log", "w")
sys.stdout = file1 # redirection
sys.stderr = file2
print("This is the output message\n") # print on file1
sys.stderr.write("This is the error message\n") # print on file2
sys.stdout = save_stdout # redirection to original streams
sys.stderr = save_stderr
file1.close()
file2.close()
```



`sys.stderr.write()` e  
`sys.stdout.write()` per  
scrivere sull std output e std error

# Modulo sys

- Parametri interessanti:
  - `sys.executable` path dell'interprete Python
  - `sys.maxint` valore **max** per gli interi
  - `sys.modules` dizionario contenente tutti i moduli caricati dall'interprete
  - `sys.path` lista di cartelle in cui Python cerca i moduli.
  - `sys.platform` nome del sistema operativo
  - ...

# Modulo NumPy

- NumPy è il package fondamentale per scienziati che programmano con Python. Tra le altre cose, contiene un potente array N-dimensionale di oggetti:
  - funzioni sofisticate (broadcasting)
  - strumenti per l'integrazione di codice C/C++ e Fortran
  - utili funzioni di algebra lineare, trasformata di Fourier e numeri casuali
  - Parte del framework SciPy (<https://www.scipy.org/>)
- Vedi <https://www.numpy.org/> per documentazione e tutorial