



**University
of Ferrara**

Python

Funzioni, moduli e classi

Funzioni

- La definizione di una funzione è:

```
def function_name(arguments) :  
    # istruzione
```

- **def** è una parola chiave di Python
- **Arguments** possono essere 0 o più variabili.
- Il corpo di una funzione (istruzione) dev'essere **indentato**.
- Se il blocco contiene la parola chiave **return** seguita da un'espressione, allora essa restituisce un valore; in alternativa restituirà il valore speciale **None**.

Funzioni

- La definizione di una funzione:

```
def f(x=10) :  
    print(x)
```

`f()`

`f(5)`

- Come argomenti può avere 0 o più variabili. Le variabili possono avere un valore di default, nel caso in cui non venissero passati parametri in ingresso.
- Il programma stamperà 10 e 5.

Funzioni

- Il nome di una funzione :
 - Deve iniziare con una lettera oppure un trattino basso
 - Può includere solo lettere, trattini bassi e numeri
- Come per le variabili, la convenzione di denominazione richiede di utilizzare solo lettere minuscole e dividere le parole con il trattino basso.

Funzioni

- E' possibile passare delle chiamate ad altre funzioni.

`sum(divide(a,b),multiply(c,d))`

- L'espressione è valutata in direzione dall'interno verso l'esterno.
- Variabili locali e globali:
 - Le variabili locali sono definite all'interno delle funzioni, le variabili globali sono definite all'esterno delle funzioni.
 - Prima, si verificano le variabili locali definite nella funzione.
 - Poi si verificano le variabili globali.
 - Infine si verificano le variabili built-in.

Aggiunta sulla definizione di funzione

- Abbiamo già visto che nella definizione di una funzione è possibile definire un valore di default per il suo argomento.

```
def useless_f(x, y, n=5, s='boh') :  
    print(x, y, n, s)
```

- Questa funzione può essere chiamata in vari modi:
 - Passando solo i parametri obbligatori **useless_f(1,2)**
 - Fornendo solo un parametro opzionale **useless_f(1,2,3)**
 - Dando tutti gli argomenti **useless_f(1,2,3,4)**
 - ... ed altro

Devono seguire l'ordine

Argomenti di default

- E' necessario porre attenzione agli argomenti di default

```
i = 5
```

```
def f(arg=i):  
    print(arg)
```

```
i = 6
```

```
f()
```

L'output sarà 5

Argomenti di default

- E' necessario porre attenzione agli argomenti di default mutabili.
- I valori default vengono valutati solo una volta. Questo fa la differenza quando l'argomento di default è un oggetto mutabile come liste, dizionari oppure, in molti casi, l'istanza di una classe.
- In Python, quando si passa un valore mutabile come argomento di default di una funzione, l'argomento viene modificato tutte le volte che l'oggetto viene modificato.
- `def f(a, L=[]):`
 `L.append(a)`
 `return L`

```
print(f(1))  The output will be [1]
print(f(2))  The output will be [1,2]
print(f(3))  The output will be [1,2,3]
```


Argomenti di default

- Dobbiamo fare attenzione agli argomenti **mutabili**.
- Se vogliamo mantenere ogni chiamata indipendente dalle precedenti, dobbiamo modificare la funzione.

```
def f(a, L=None):  
    if L is None:  
        L = []  
    L.append(a)  
    return L
```

None è immutabile,
ogni modifica crea
un nuovo valore

is al momento è
equivalente a **==**

Aggiunta sulla definizione di funzione

```
def useless_f(x, y, n=5, s='boh') :  
    print(x, y, n, s)
```

- Questa funzione può essere chiamata in vari modi:
 - Passando solo i parametri obbligatori `useless_f(1,2)`
 - Fornendo solo un parametro opzionale `useless_f(1,2,3)`
 - Dando tutti gli argomenti `useless_f(1,2,3,4)`
 - Passando coppie chiave valore `useless_f(s=3,x=1,y=2,n=4)`

In questo modo non è necessario seguire l'ordine degli argomenti.
Possiamo fornire tutti o solo alcuni tra i valori opzionali.

ATTENZIONE: `useless_f(1,2,n=4)` è valido
`useless_f(1,2,s=3,4)` **non** lo è

Funzioni con un numero variabile di argomenti (liste e mappe)

- Prendete questa definizione di funzione:

```
def function_name(arg1, arg2, *arg1, **argm) :  
    # istruzione
```

- ***arg1** è una lista di lunghezza arbitraria
- ****argm** è un dizionario di lunghezza arbitraria

```
function_name(1, 2, 3, 4, val1=5, val2=6)
```

Dopo questa chiamata **arg1=[3, 4]** e **argm={ 'val1' : 5, 'val2' : 6 }**

Funzioni con un numero variabile di argomenti (liste e mappe)

- Il * può essere usato anche per passare dei valori alla funzione

```
def f(x, y):  
    pass # to impl. after
```

```
l = [1, 2]
```

```
f(*l)
```

Assegna 1 a **x** e 2 a **y**.

L'istruzione pass non fa nulla. Può essere usata quando è richiesta un'istruzione per correttezza sintattica, ma il programma non richiede un'azione.

Funzioni con un numero variabile di argomenti (liste e mappe)

- Il `*` può essere usato anche per passare dei valori a una funzione

```
def f(x,y):  
    pass # to impl. after
```

```
l = [1,2]
```

```
f(*l)
```

Assegna 1 a `x` e 2 a `y`.

```
l = [1,2]
```

```
range(*l)
```

Può essere usata con tutte le funzioni. Anche built-in.

Funzioni con un numero variabile di argomenti (liste e mappe)

- Il `**` può essere usato anche per passare dei valori a una funzione

```
def f(x, y):  
    pass # to impl. after
```

```
m = {'y':1, 'x':2}
```

```
f(**m)
```

Assegnerà 2 a `x` e 1 a `y`.

L'ordine nel dizionario non è importante.

Documentazione sulle funzioni

- La funzione integrata `help()` può essere usata per ottenere informazioni su una particolare funzione o modulo.
- Se invochiamo `help()` con una funzione che abbiamo definite noi, otterremo le informazioni scritte nella `docstring` (se l'abbiamo scritta).
- Una `docstring` è delimitate da `' ' '` e dev'essere la prima linea del modulo o della funzione.

Documentazione sulle funzioni

Inizio sulla prima
riga

```
def f(x=10):
```

```
'''
```

```
    this function does nothing
```

```
'''
```

```
    print(x)
```

Nota
l'indentazione

Convenzione: lascia
vuota la linea dopo il
docstring

```
help(f)
```

- La funzione help() stamperà

```
f(x=10)
```

```
    this function does nothing
```


Moduli

- I moduli consentono di definire gruppi di funzioni e variabili tra loro in relazione.
- Il nome del modulo è il nome del file senza **.py**
- Per usare un modulo, è necessario importarlo (**import**).
 - NOTA:
 - L'importazione di un modulo fa sì che Python esegua ogni riga di codice nel modulo, ovvero, se ci sono operazioni al di fuori delle funzioni, vengono eseguite.
 - Per richiamare una funzione del modulo: **module_name.function_name()**

```
import my_module  
my_module.function_name()
```

Moduli

- Ogni modulo ha una variabile implicita `__name__`.
- Se importiamo un modulo chiamato `module_m`, allora
`module_m.__name__ == "module_m"`
- Ma se eseguiamo un modulo, allora
`__name__ == "__main__"`
- Ricordiamo che se eseguiamo un modulo, non abbiamo bisogno del nome come prefisso.
- Per verificare se un modulo è in esecuzione possiamo definire una sorta di funzione main nel modulo che verrà svolta solo se il modulo è in esecuzione.

Moduli

a.py

```
import b

def f(x):
    return x * 2

if __name__ == "__main__":
    print(1)
    print(b.f(10))
    print(f(10))
```

b.py

```
import a

def f(x):
    return x / 2

if __name__ == "__main__":
    print(1)
    print(a.f(10))
    print(f(10))
```

Moduli

```
from module_name import fn_name1, fn_name2
```

- Per importare solo le funzioni **fn_name1** e **fn_name2**
- Le funzioni importate possono essere richiamate usando il loro nome
- E' possibile usare * per importare tutte le funzioni all'interno del modulo.

```
from module_name import *
```

- Se due moduli hanno funzioni con lo stesso nome, rimarranno solo quelle più recenti.

Moduli

a.py

```
import b

def f(x):
    return x * 2

if __name__ == "__main__":
    print(1)
    print(b.f(10))
    print(f(10))
```

b.py

```
from a import f

def f(x):
    return x / 2

if __name__ == "__main__":
    print(1)
    print(a.f(10))
    print(f(10))
```

Moduli

a.py

```
import b
```

```
def f(x):  
    return x * 2
```

b.py

```
from a import f
```

```
def f(x):  
    return x / 2
```

```
if
```

Traceback (most recent call last):

File "/Users/fabrizio/Downloads/b.py", line 1, in <module>

from a import f

File "/Users/fabrizio/Downloads/a.py", line 1, in <module>

import b

File "/Users/fabrizio/Downloads/b.py", line 1, in <module>

from a import f

ImportError: cannot import name 'f' from partially initialized module 'a' (most likely due to a circular import)

(/Users/fabrizio/Downloads/a.py)

Ancora sui moduli

- Importare un modulo significa creare una variabile con il nome del modulo e che contiene il riferimento al modulo stesso.

Ancora sui moduli

- Importare un modulo significa creare una variabile che abbia il nome del modulo e contenga il riferimento al modulo stesso.
- Possiamo usare **as** per rinominare una variabile.

```
import a as module_a  
module_a.f(10)
```

```
from a import f as func  
func(10)
```


Ancora sui moduli

- Il `from import as` può essere simulato anche assegnando il riferimento della funzione ad una variabile.

```
func=a.f
```

```
func(10)
```

Esercizio 5

1. Creare due moduli Python chiamati rispettivamente `module_one` e `module_two`.
2. `module_one` definisce le funzioni:
 1. `f` prende in ingresso 1 o 2 argomenti `x` e `y`, con `y` definita di default pari a 10, e restituisce la loro somma.
 2. `g` prende un argomento `x` e calcola x^3 (usa l'esponenziale)
3. Inoltre `module_one` definisce un `main` che chiama `f(2)` e `f(g(2))` e stampa i risultati.
4. `module_two` importa solo la funzione `f` da `module_one` e definisce un `main` che chiama `f(10, 5)` stampando il risultato.

Esercizio 5

5. Eseguire prima un modulo poi l'altro

Funzioni vs Metodi

- I metodi sono simili alle funzioni ma sono collegati al tipo dell'oggetto.
- Possono essere richiamati a partire da un oggetto usando la notazione puntata.

```
>>> s = "Python"
```

```
>>> s.upper() # questo metodo restituisce "PYTHON"
```

```
>>> s.lower() # questo metodo restituisce "python"
```

Funzioni vs Metodi

- Le funzioni appartengono ad un modulo.
- I metodi appartengono ad un oggetto

- `len(str)` è una funzione
- `str.lower()` è un metodo.

Classi

- Oltre ai moduli e alle funzioni, Python consente di definire le classi, seguendo il paradigma di programmazione orientata agli oggetti.
- In Python una classe è un nuovo tipo, usando una classe possiamo istanziare oggetti della classe.

Classi

- Una classe è definita dalla parola chiave **class** come segue:

```
class MyClass:  
    # definizioni
```

Classi

- Una classe è definita dalla parola chiave **class** come segue :

```
class MyClass:  
    # definizione
```

- Convenzione sui nomi delle classi: si usa **UpperCamelCase**

Classi

- Dimentichiamoci per un momento dell'ereditarietà, una volta definita una classe:

```
class MyClass:  
    # definizione
```

possiamo creare un'istanza come segue:

```
x = MyClass()
```

Classi

- La chiamata di `x = MyClass()` crea un nuovo oggetto vuoto.
- Se vogliamo inizializzare l'attributo di una classe dobbiamo specificare il costruttore.
- Possiamo usare il metodo `__init__()` che viene automaticamente invocato alla creazione di un oggetto.
- E' possibile passare dei valori al metodo `__init__()`
 - ATTENZIONE: può essere definito solo un costruttore. L'overloading in Python è fatto utilizzando i valori di default degli argomenti.

Classi: un esempio

```
class Point:  
    def __init__(self, x = 0, y = 0):  
        self.x = x  
        self.y = y
```

- Questa classe definisce punti nello spazio 2D per mezzo di 2 attributi, chiamati **x** e **y**.
- Nella definizione dei metodi, dobbiamo passare **self** come primo argomento per assicurarci che le operazioni vengano eseguite sulla specifica istanza.

Classi: un esempio

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

- L'utilizzo di argomenti di default simula l'implementazione di più costruttori.

```
p1 = Point()          # senza parametri in ingresso
p2 = Point(1)          # passando x
p3 = Point(1, 2)       # passando x ed y
p4 = Point(y = 2)      # passando solo y
```

Classi: un esempio

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y
```

- L'utilizzo di argomenti di default e il primo attributo consente la creazione di più costruttori.

```
p1 = Point()          # senza argomenti
p2 = Point(1)         # passando solo x
p3 = Point(1, 2)      # passando x ed y
p4 = Point(y = 2)     # passando solo y
```

Nota che l'uso di self come primo attributo consente la possibilità di non passare alcun parametro al costruttore.

Classi

- Abbiamo già visto metodi che lavorano su istanze.
- La definizione di metodo è simile alla definizione di funzione. Una delle differenze è che il metodo dev'essere definito all'interno della definizione di una classe.
- Inoltre, sappiamo che i metodi lavorano sui valori che definiscono lo stato di un'istanza → le classi hanno degli **attributi** che definiscono il loro stato.
- E' possibile accedere ad un attributo utilizzando la notazione con il punto → **p1.x**

Metodi

- La definizione di un metodo è:

```
def method_name(self, arguments) :  
    # istruzioni
```

- Dev'essere fatta all'interno di una classe
- **arguments** può essere 0 o più variabili
- Il corpo della funzione (istruzioni) dev'essere **indentato**.
- Se il blocco contiene la parola chiave **return** seguita da un'espressione, allora il metodo restituisce un valore; altrimenti restituisce il valore **None**.
- Per riferirsi agli attributi dell'oggetto stesso si usa la parola **self**.

Classi: un esempio

```
class Point:
    '''
    This is a 2-D point
    Attributes: x,y
    '''

    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def distance_from_origin(self):
        return ((self.x**2) +
                (self.y**2))**(1/2)
```


Metodi

- Dobbiamo aggiungere come primo argomento di ciascun metodo il riferimento **self**.
- Questo è dovuto al fatto che Python vede ciascun metodo come il metodo di una classe.
- Un metodo può essere invocato:
 - Usando la notazione con il punto sull'istanza
 - Usando la notazione a punto sul nome della classe, passando un'istanza.

inst.method() → **Class.method(inst)**

Questi due modalità sono tra loro identiche. Python automaticamente associa **inst** a **self** (Il primo argomento del metodo)

Metodi Speciali

- `__` indica che il metodo è un metodo speciale.
- Questo viene utilizzato per rendere la nostra classe più simile ad un tipo built-in di Python.
- Per esempio:
 - `__str__` è usato per stampare (converte lo stato di un oggetto in una stringa).
 - `__cmp__` è usato per consentire l'uso di operazioni di confronto
 - `__cmp__(self, other)` restituisce -1 se **self** è "più piccolo" di **other**, 0 se sono uguali e 1 altrimenti.
 - `__eq__`, `__lt__`, `__gt__`
 - `__add__` è usato per consentire l'operazione +
 - `__iter__` è usato per consentire al nuovo tipo di essere usato in un ciclo.
 - `__doc__` restituisce una docstring

Attributi Speciali

- La variabile `__class__` contiene il nome della classe

```
def __comp__(self, other):  
    if other.__class__ is self.__class__:  
        # confronta con gli altri  
        # attributi
```

Convenzioni sulle classi

- Il nome delle classi usa la convenzione **UpperCamelCase**
- I metodi e le istanze delle classi iniziano con la lettera minuscola e hanno le parole separate tramite trattino basso
- Le definizioni dei metodi possono avere delle docstrings proprio come le definizioni delle funzioni.
- Le classi dovrebbero aver docstrings, proprio come i moduli, per descrivere cosa fanno.

Oggetti

- Abbiamo visto che ci sono oggetti che sono immutabili (int, str, etc.)
- Come le stringhe in Java, se noi proviamo a modificarli andiamo a creare una nuova istanza ed ad eliminare l'istanza precedente.
 - Questo comportamento funziona bene con gli oggetti semplici ma non con quelli più complessi che sono la maggioranza.
 - Abbiamo bisogno di **oggetti mutabili**

Oggetti mutabili

- Se vogliamo cambiare un oggetto molto ampio senza mantenere l'originale, allora copiarlo, modificare la copia e poi eliminare l'originale, è dispendioso e poco utile.
- Di contro, possiamo usare un oggetto mutabile che possiamo quindi cambiare a nostro piacimento.
- Questo ci consente di definire funzioni che modificano l'oggetto passato piuttosto che crearne ogni volta uno da zero.

Aliasing

```
x=10
```

```
y=x
```

```
x=5
```

```
print(x, y)
```

- Sappiamo che questo stamperà a video **5 10**, perchè gli interi sono immutabili

```
x=my_mutable_obj
```

```
y=x
```

```
x.chg_something()
```

```
print(x, y)
```

- Questo stamperà **x** due volte

Aliasing

```
x=10
```

```
y=x
```

```
x=5
```

```
print(x, y)
```

- Sappiamo che questo stamperà a video **5 10**, perchè gli interi sono immutabili

x ed **y** prima puntano allo stesso oggetto, poi **x** viene modificato e così punterà ad un nuovo oggetto.

```
mutable_obj
```

```
_something()
```

```
print(x, y)
```

- Questo stamperà **x** due volte

Aliasing

```
x=10
```

```
y=x
```

```
x=5
```

```
print(x, y)
```

x ed **y** prima puntano allo stesso oggetto, poi **x** viene modificato e di conseguenza anche **y** visto che non viene creato un nuovo oggetto

- Sappiamo che questo stamperà a video **5 10**, perchè gli interi sono immutabili

```
x=my_mutable_obj
```

```
y=x
```

```
x.chg_something()
```

```
print(x, y)
```

- Questo stamperà **x** due volte

Aliasing e funzioni

- Quando si chiama una funzione, si inizia effettivamente con una serie di istruzioni di assegnazione.
- Cioè, i valori dei parametri effettivi sono assegnati alle variabili locali.
 - Per gli oggetti immutabili, le modifiche punteranno a nuovi oggetti, quindi nel blocco di istruzioni chiamante queste modifiche non verranno visualizzate
 - Ma con gli oggetti mutabili, queste istruzioni di assegnazione indicano che la variabile locale fa riferimento a un oggetto mutabile che può cambiare.
 - Questo è il motivo per cui le funzioni possono cambiare gli oggetti mutabili, ma non quelli immutabili.

Aliasing e funzioni

```
def f(x,y):  
    x = x.lower()  
    y.chg_something()  
    print(x)
```

```
x = "Python"  
y = my_mutable_obj  
f(x,y)  
print(x)  
print(y)
```

- **y** è cambiato nella funzione, stampandolo noi vedremo questo cambiamento.
- **x** non è cambiato nella funzione. Visto che la funzione non restituisce **x**, la stampa di **x** mostrerà **"Python"** e non **"python"**

Attributi di classi

- La gestione degli attributi in Python non è banale. Un attributo in una classe può essere definito in due modi:
 - All'interno di una classe ma al di fuori di qualunque metodo
 - All'interno di un metodo

```
class Dog:
    kind = 'canine'
    def __init__(self, name):
        self.name = name
```

- La variabile `kind` è condivisa tra le istanze (attributo di classe - come le variabili statiche in Java)
- La variabile `name` è unica in ciascuna istanza (attributo di istanza - come le variabili non statiche in Java)

Attributi di classi

- Possiamo accedere agli attributi di classe in due modi:
- Accesso nei metodi di classe utilizzando il parametro `self` o il nome della classe (`self.kind` o `Dog.kind`)
- Accesso dall'esterno utilizzando un'istanza o il nome della classe (`fido.kind` o `Dog.kind`)

Attributi di classi

```
class Dog:
    kind = 'canine'
    def __init__(self, name):
        self.name = name
    def is_old(self, age):
        self.age = age
```

canine, fido

canine, doggo

pet, fido

pet, doggo

pet, fido, 3

AttributeError: 'Dog' object has no attribute 'age'

```
d=Dog('fido')
e=Dog('doggo')
print(d.kind, d.name)
print(e.kind, e.name)
Dog.kind='pet'
print(d.kind, d.name)
print(e.kind, e.name)
d.is_old(3)
print(d.kind, d.name, d.age)
print(e.kind, e.name, e.age)
```

Attributi di classi

```
class Dog:
    kind = 'canine'
    def __init__(self, name):
        self.name = name
    def is_old(self, age):
        self.age = age
```

```
canine, fido
```

```
canine, doggo
```

```
canine, fido
```

```
pet, doggo
```

```
canine, fido, 3
```

```
AttributeError: 'Dog' object has no attribute 'age'
```

```
d=Dog('fido')
e=Dog('doggo')
print(d.kind, d.name)
print(e.kind, e.name)
e.kind='pet'
print(d.kind, d.name)
print(e.kind, e.name)
d.is_old(3)
print(d.kind, d.name, d.age)
print(e.kind, e.name, e.age)
```

Attributi di classi

```
class Dog:
```

```
    kind = 'canine'
```

```
    def __init__(self, name):
```

```
        self.name = name
```

Si comporta in questo modo perché `e.kind='pet'` in realtà crea un nuovo attributo di istanza che si chiama `kind` che maschera quello di classe

```
canine, doggo
```

```
canine, fido
```

```
pet, doggo
```

```
canine, fido, 3
```

```
AttributeError: 'Dog' object has no attribute 'age'
```

```
d=Dog('fido')
```

```
e=Dog('doggo')
```

```
print(d.kind,d.name)
```

```
print(e.kind,e.name)
```

```
e.kind='pet'
```

```
print(d.kind,d.name)
```

```
print(e.kind,e.name)
```

```
d.is_old(3)
```

```
print(d.kind,d.name,d.age)
```

```
print(e.kind,e.name,e.age)
```


Attributi di classi

```
class Dog:
    toys = []
    def __init__(self,name):
        self.name = name
    def add_toy(self,toy):
        self.toys.append(toy)
```

```
fido, []
doggo, []
fido, [red ball, peluches]
doggo, [red ball, peluches]
```

```
d=Dog('fido')
e=Dog('doggo')
print(d.name,d.toys)
print(e.name,e.toys)
d.add_toy('red ball')
e.add_toy('peluches')
print(d.name,d.toys)
print(e.name,e.toys)
```

Attributi di classi: corretto

```
class Dog:
    def __init__(self, name):
        self.name = name
        self.toys = []
    def add_toy(self, toy):
        self.toys.append(toy)
```

```
fido, []
doggo, []
fido, [red ball]
doggo, [peluches]
```

```
d=Dog('fido')
e=Dog('doggo')
print(d.name,d.toys)
print(e.name,e.toys)
d.add_toy('red ball')
e.add_toy('peluches')
print(d.name,d.toys)
print(e.name,e.toys)
```

Basato su slide del Prof. Riccardo Zese
riccardo.zese@unife.it