Programmazione avanzata a.a. 2024-25

A. De Bonis

Introduzione a Python (III parte)

1

Funzioni in Python

- Le funzioni sono definite usando la keyword **def**
- Viene introdotto un nuovo identificatore (il nome della funzione)
- Devono essere specificati
  - Il nome e la lista dei parametri
  - La funzione può avere un numero di parametri variabile
- L'istruzione **return** (opzionale) restituisce un valore ed interrompe l'esecuzione della funzione

2

Esempi

```
def contains(data, target):
    for item in data:
        if item == target:
            return True
    return False
```

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

```
def sum(values):
  total = 0
  for v in values:
    total = total + v
  return total
```


3

Esempi

```
def bubble_sort(a):
    n=len(a)
    while(n>0):
        for i in range(0,n-1):
            if(a[i]>a[i+1]):
                a[i], a[i+1] = a[i+1], a[i]
        n -= 1
    return a
```

Assegnamento multiplo
swap in un rigo

```
a = [5, 3, 1, 7, 8 ,2]
print(a)
bubble_sort(a)
print(a)
```

↓

```
[5, 3, 1, 7, 8, 2]
[1, 2, 3, 5, 7, 8]
```

Il parametro a è passato per riferimento

```
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = bubble_sort(a[:])
print('b =', b)
print('a =', a)
```

↓

```
a = [5, 3, 1, 7, 8, 2]
b = [1, 2, 3, 5, 7, 8]
a = [5, 3, 1, 7, 8, 2]
```


4

Stringa di documentazione

- La prima riga di codice nella definizione di una funzione dovrebbe essere una breve spiegazione di quello che fa la funzione
  - docstring

```
def my_function():
    """Do nothing, but document it. ...

        No, really, it doesn't do anything.
    """

    pass # Istruzione che non fa niente
```

```
print(my_function.__doc__)
```

↓

```
Do nothing, but document it. ...

        No, really, it doesn't do anything.
```


5

Variabili globali

- Nel corpo di una funzione si può far riferimento a variabili definite nell'ambiente (scope) esterno alla funzione, ma tali variabili non possono essere modificate
- Per poterle modificare bisogna dichiararle **global** nella funzione
  - Se si prova ad accedere ad esse senza dichiararle global viene generato un errore

6

Esempi

```
n = 111
def f():
   print('nella funzione n =', n)

f()
print('fuori la funzione n =', n)
```

→

```
nella funzione n = 111
fuori la funzione n = 111
```

```
m = 999
def f1():
   m = 1
   print('nella funzione m =', m)

f1()
print('fuori la funzione m =', m)
```

→

```
nella funzione m = 1
fuori la funzione m = 999
```


7

Esempi

```
m=999
def f3():
   print('nella funzione m =', m)
   m = 1

f3()
print('fuori la funzione m =', m)
```

↘

```
UnboundLocalError: local variable 'm' referenced before assignment
```

```
n = 777
def varGlobaliQuattro():
   global n
   print('nella funzione n =', n)
   n=3

print('fuori la funzione n =', n)
varGlobaliQuattro()
print('fuori la funzione n =', n)
```

→

```
fuori la funzione n = 777
nella funzione n = 777
fuori la funzione n = 3
```


8

Parametri di una funzione

- Parametri formali di una funzione
  - Identificatori usati per descrivere i parametri di una funzione nella sua definizione
- Parametri attuali di una funzione
  - Valori passati alla funzione all'atto della chiamata
  - Argomenti di una funzione
- Argomento keyword
  - Argomento preceduto da un identificatore in una chiamata a funzione
- Argomento posizionale
  - Argomento che non è un argomento keyword

9

Passaggio dei parametri

- Il passaggio dei parametri avviene tramite un riferimento ad oggetti
  - Per valore, dove il valore è il riferimento (puntatore) dell'oggetto passato

```
lst = [1, 'due']


def modifica(lista):
    lista.append('nuovo')


print('lista =', lst)
modifica(lst)
print('lista =', lst)
```



```
lista = [1, 'due']

lista = [1, 'due', 'nuovo']
```


10

Parametri di default

- Nella definizione della funzione, ad ogni parametro formale può essere assegnato un valore di default
  - **a partire da quello più a destra**
- La funzione può essere invocata con un numero di parametri inferiori rispetto a quello con cui è stata definita

```
def default(a, b=3):
    print('a =', a, 'b =', b)


default(2)
default(1,1)
```

→

```
a = 2 b = 3
a = 1 b = 1
```


11

Parametri di default

- Gli argomenti di default devono sempre seguire quelli non di default.
  - **la funzione f nel riquadro è definita in modo sbagliato**

```
>>> def f(a=1,b):
...     print(a,b)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```


12

Attenzione

- I parametri di default sono valutati nello scope in cui è definita la funzione

```
d = 666
def default_due(a, b=d):
   print('a =', a, 'b =', b)

d = 0
default_due(11)
default_due(22,33)
```

→

```
a = 11 b = 666
a = 22 b = 33
```


13

Attenzione

- I parametri di default sono valutati solo una volta (quando si definisce la funzione)
  - **Attenzione a quando il parametro di default è un oggetto mutable**

```
def f(a, L=[]):
   L.append(a)
   return L

print(f(1))
print(f(2))
print(f(3))
```

→

```
[1]
[1, 2]
[1, 2, 3]
```

La lista L conserva il proprio valore tra chiamate successive, non è inizializzata ad ogni chiamata

14

Attenzione

- Se non si vuole che il parametro di default sia condiviso tra chiamate successive si può adottare la seguente tecnica (lo si inizializza nel corpo della funzione)

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

print(f(1))
print(f(2))
print(f(3))
```



```
[1]
[2]
[3]
```


Numero variabile di argomenti

- In Python si possono definire funzioni con un numero variabile di parametri
- L'ultimo parametro è preceduto da *
- Dopo ci possono essere solo parametri keyword (dettagli in seguito)
- Il parametro formale preceduto da * indica la sequenza in cui sono contenuti un numero variabile di parametri
  - Nel corpo della funzione possiamo accedere al valore di questi parametri tramite la posizione

Esempio

```
def variabili(v1, v2=4, *arg):
    print('primo parametro =', v1)
    print('secondo parametro =', v2)
    print('# argomenti passati', len(arg) + 2)
    if arg:
        print('# argomenti variabili', len(arg))
        print('arg =', arg)
        print('primo argomento variabile =', arg[0])
    else:
        print('nessun argomento in più')
```

variabili(1, **'a'**, 4, 5, 7)

```
primo parametro = 1
secondo parametro = a
# argomenti passati 5
# argomenti variabili 3
arg = (4, 5, 7)
primo argomento variabile = 4
```

variabili(3, **'b'**)

```
primo parametro = 3
secondo parametro = b
# argomenti passati 2
nessun argomento in più
```


17

L'operatore *

- Ogni tipo iterabile può essere spacchettato usando l'operatore * (unpacking operator).
- Se in un assegnamento con due o più variabili a sinistra dell'assegnamento, una di queste variabili è preceduta da * allora i valori a destra sono assegnati uno ad uno alle variabili (senza *) e i restanti valori vengono assegnati alla variabile preceduta da *.
- Possiamo passare come argomento ad una funzione che ha k parametri posizionali una collezione iterabile di k elementi preceduta da *
  - Questo è diverso dal caso in cui utilizziamo * davanti ad un parametro formale

18

Esempi di uso di *

```
>>> primo, secondo, *rimanenti = [1,2,3,4,5,6]
>>> primo
1
>>> secondo
2
>>> rimanenti
[3, 4, 5, 6]
```

```
>>> primo, *rimanenti, sesto, = [1,2,3,4,5,6]
>>> primo
1
>>> sesto
6
>>> rimanenti
[2, 3, 4, 5]
```


19

Esempi di uso di *

```
def variabili(v1, v2=4, *arg):
    print('primo parametro =', v1)
    print('secondo parametro =', v2)
    print('# argomenti passati', len(arg) + 2)
    if arg:
        print('# argomenti variabili', len(arg))
        print('arg =', arg)
        print('primo argomento variabile =', arg[0])
    else:
        print('nessun argomento in più')
```

variabili(1, 'a', 4, 5, 7)

L=[4,5,7]
variabili(1,'a',*L)

```
primo parametro = 1
secondo parametro = a
# argomenti passati 5
# argomenti variabili 3
arg = (4, 5, 7)
primo argomento variabile = 4
```


20

Esempi di uso di *

```
def somma(addendo1, addendo2, addendo3):
    return addendo1+addendo2+addendo3

addendi=[56,2,4]

print("somma =",somma(*addendi))
```

Attenzione:
addendi deve contenere esattamente 3 elementi

```
somma = 62
```


21

Unpacking

- Quando a sinistra di un assegnamento ci sono due o più variabili e a sinistra c'è una sequenza, la collezione viene spacchetata e gli elementi assegnati alle variabili a sinistra
  - Lo abbiamo già visto per le tuple
- Esempio:

```
>>> l=[1,2,3,4]
>>> a,b,c,d = l
>>> a
1
>>> b
2
>>> c
3
>>> d
4
```


22

Solo per avere un'idea di cosa si puo` fare con reflection

```
>>> def g():
...     print("sono nella funzione g")
...
>>> def f():
...     f.__code__=g.__code__
...     print("e` stata invocata la versione non modificata")
...
>>> f()
e` stata invocata la versione non modificata
>>> f()
sono nella funzione g
```

la funzione originaria puo` essere eseguita una sola volta!

Esempi

```
def contains(data, target):
    for item in data:
        if item == target:
            return True
    return False
```

```
def count(data, target):
  n = 0
  for item in data:
    if item == target:
      n += 1
  return n
```

```
def sum(values):
  total = 0
  for v in values:
    total = total + v
  return total
```

Esempi

```
def bubble_sort(a):
    n=len(a)
    while(n>0):
        for i in range(0,n-1):
            if(a[i]>a[i+1]):
                a[i], a[i+1] = a[i+1], a[i]
        n -= 1
    return a
```

Assegnamento multiplo
swap in un rigo

```
a = [5, 3, 1, 7, 8 ,2]
print(a)
bubble_sort(a)
print(a)
```

```
[5, 3, 1, 7, 8, 2]
[1, 2, 3, 5, 7, 8]
```

Il parametro a è passato per riferimento

```
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = bubble_sort(a[:])
print('b =', b)
print('a =', a)
```

```
a = [5, 3, 1, 7, 8, 2]
b = [1, 2, 3, 5, 7, 8]
a = [5, 3, 1, 7, 8, 2]
```


25

Stringa di documentazione

- La prima riga di codice nella definizione di una funzione dovrebbe essere una breve spiegazione di quello che fa la funzione
  - docstring

```
def my_function():
    """Do nothing, but document it. ...

        No, really, it doesn't do anything.
    """
    pass # Istruzione che non fa niente
```

```
print(my_function.__doc__)
```

```
Do nothing, but document it. ...

    No, really, it doesn't do anything.
```


26

Variabili globali

- Nel corpo di una funzione si può far riferimento a variabili definite nell'ambiente (scope) esterno alla funzione, ma tali variabili non possono essere modificate
- Per poterle modificare bisogna dichiararle **global** nella funzione
  - Se si prova ad accedere ad esse senza dichiararle global viene generato un errore

Esempi

```
n = 111
def f():
   print('nella funzione n =', n)

f()
print('fuori la funzione n =', n)
```

→

```
nella funzione n = 111
fuori la funzione n = 111
```

```
m = 999
def f1():
   m = 1
   print('nella funzione m =', m)

f1()
print('fuori la funzione m =', m)
```

→

```
nella funzione m = 1
fuori la funzione m = 999
```

Esempi

```
m=999
def f3():
   print('nella funzione m =', m)
   m = 1

f3()
print('fuori la funzione m =', m)
```

UnboundLocalError: local variable 'm' referenced before assignment

```
n = 777
def varGlobaliQuattro():
   global n
   print('nella funzione n =', n)
   n=3

print('fuori la funzione n =', n)
varGlobaliQuattro()
print('fuori la funzione n =', n)
```

fuori la funzione n = 777
nella funzione n = 777
fuori la funzione n = 3

29

Parametri di una funzione

- Parametri formali di una funzione
  - Identificatori usati per descrivere i parametri di una funzione nella sua definizione
- Parametri attuali di una funzione
  - Valori passati alla funzione all'atto della chiamata
  - Argomenti di una funzione
- Argomento keyword
  - Argomento preceduto da un identificatore in una chiamata a funzione
- Argomento posizionale
  - Argomento che non è un argomento keyword: l'associazione tra parametro formale e parametro attuale è stabilita dalla posizione.

30

Passaggio dei parametri

- Il passaggio dei parametri avviene tramite un riferimento ad oggetti
  - Per valore, dove il valore è il riferimento (puntatore) dell'oggetto passato

```
lst = [1, 'due']


def modifica(lista):
    lista.append('nuovo')


print('lista =', lst)
modifica(lst)
print('lista =', lst)
```

→

```
lista = [1, 'due']

lista = [1, 'due', 'nuovo']
```


31

Parametri di default

- Nella definizione della funzione, ad ogni parametro formale può essere assegnato un valore di default
  - **a partire da quello più a destra**
- La funzione può essere invocata con un numero di parametri inferiori rispetto a quello con cui è stata definita

```
def default(a, b=3):
    print('a =', a, 'b =', b)


default(2)
default(1,1)
```

→

```
a = 2 b = 3
a = 1 b = 1
```


32

Parametri di default

- Gli argomenti di default devono sempre seguire quelli non di default.
  - **la funzione f nel riquadro è definita in modo sbagliato**

```
>>> def f(a=1,b):
...     print(a,b)
...
  File "<stdin>", line 1
SyntaxError: non-default argument follows default argument
```


Attenzione

- I parametri di default sono valutati nello scope in cui è definita la funzione

```
d = 666
def default_due(a, b=d):
    print('a =', a, 'b =', b)

d = 0
default_due(11)
default_due(22,33)
```

→

```
a = 11 b = 666
a = 22 b = 33
```

Attenzione

- I parametri di default sono valutati solo una volta (quando si definisce la funzione)
  - **Attenzione a quando il parametro di default è un oggetto mutable**

```
def f(a, L=[]):
    L.append(a)
    return L

print(f(1))
print(f(2))
print(f(3))
```

La lista L conserva il proprio valore tra chiamate successive, non è inizializzata ad ogni chiamata

```
[1]
[1, 2]
[1, 2, 3]
```


35

Attenzione

- Se non si vuole che il parametro di default sia condiviso tra chiamate successive si può adottare la seguente tecnica (lo si inizializza nel corpo della funzione)

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

print(f(1))
print(f(2))
print(f(3))
```

sono due oggetti diversi (diversi id)

```
[1]
[2]
[3]
```


36

Attenzione

- Un altro esempio:

```
def h(c,L=[]):
    print(L,id(L))
    L.append(c)
    print(L,id(L))
    L=L+['bob']
    print(L,id(L))
    return L

x1=h(100)
print("La lista restituita da h(100) e`",x1, "e ha id=",id(x1))
print("Invochiamo ora h(200)")
x2=h(200)
print("La lista restituita da h(200) e`",x2, "e ha id=",id(x2)))
```

```
[] 4338902272
[100] 4338902272
[100, 'bob'] 4339719936
La lista restituita da h(100) e` [100, 'bob'] e ha id= 4339719936
Invochiamo ora h(200)
[100] 4338902272
[100, 200] 4338902272
[100, 200, 'bob'] 4339720000
La lista restituita da h(200) e` [100, 200, 'bob'] e ha id= 4339720000
```


37

Numero variabile di argomenti

- In Python si possono definire funzioni con un numero variabile di parametri posizionali
  - In questo caso, l'ultimo parametro posizionale è preceduto da *
  - Dopo ci possono essere solo parametri keyword (dettagli in seguito)
- Il parametro formale preceduto da * indica la sequenza in cui sono contenuti un numero variabile di parametri
  - Nel corpo della funzione possiamo accedere al valore di questi parametri tramite la posizione

38

Esempio

```
def variabili(v1, v2=4, *arg):
    print('primo parametro =', v1)
    print('secondo parametro =', v2)
    print('# argomenti passati', len(arg) + 2)
    if arg:
        print('# argomenti variabili', len(arg))
        print('arg =', arg)
        print('primo argomento variabile =', arg[0])
    else:
        print('nessun argomento in più')
```

variabili(1, **'a'**, 4, 5, 7)

```
primo parametro = 1
secondo parametro = a
# argomenti passati 5
# argomenti variabili 3
arg = (4, 5, 7)
primo argomento variabile = 4
```

variabili(3, **'b'**)

```
primo parametro = 3
secondo parametro = b
# argomenti passati 2
nessun argomento in più
```


39

L'operatore *

- Ogni tipo iterabile può essere spacchettato usando l'operatore * (unpacking operator).
- Se in un assegnamento con due o più variabili a sinistra dell'assegnamento, una di queste variabili è preceduta da * allora i valori a destra sono assegnati uno ad uno alle variabili (senza *) e i restanti valori vengono assegnati alla variabile preceduta da *.
- Possiamo passare come argomento ad una funzione che ha k parametri posizionali una collezione iterabile di k elementi preceduta da *
  - Questo è diverso dal caso in cui utilizziamo * davanti ad un parametro formale per indicare un numero variabile di parametri.

40

Esempi di uso di *

```
>>> primo, secondo, *rimanenti = [1,2,3,4,5,6]
>>> primo
1
>>> secondo
2
>>> rimanenti
[3, 4, 5, 6]
```

```
>>> primo, *rimanenti, sesto, = [1,2,3,4,5,6]
>>> primo
1
>>> sesto
6
>>> rimanenti
[2, 3, 4, 5]
```


41

Esempi di uso di *

```
def variabili(v1, v2=4, *arg):
    print('primo parametro =', v1)
    print('secondo parametro =', v2)
    print('# argomenti passati', len(arg) + 2)
    if arg:
        print('# argomenti variabili', len(arg))
        print('arg =', arg)
        print('primo argomento variabile =', arg[0])
    else:
        print('nessun argomento in più')
```

variabili(1, 'a', 4, 5, 7)

L=[4,5,7]
variabili(1,'a',*L)

```
primo parametro = 1
secondo parametro = a
# argomenti passati 5
# argomenti variabili 3
arg = (4, 5, 7)
primo argomento variabile = 4
```


42

Esempi di uso di *

```
def somma(addendo1, addendo2, addendo3):
    return addendo1+addendo2+addendo3

addendi=[56,2,4]
print("somma =",somma(*addendi))
```

Attenzione:
addendi deve contenere esattamente 3 elementi

```
somma = 62
```


43

Unpacking

- Quando a sinistra di un assegnamento ci sono due o più variabili e a destra c'è una sequenza, la collezione viene spacchetata e gli elementi assegnati alle variabili a sinistra
  - Lo abbiamo già visto per le tuple
- Esempio:

```
>>> l=[1,2,3,4]
>>> a,b,c,d = l
>>> a
1
>>> b
2
>>> c
3
>>> d
4
```


44

Parametri keyword

- Sono parametri il cui valore è determinato assegnando un valore ad una keyword (nome =) oppure passato come valore (associato ad una keyword) all'interno di un dizionario (**dict**) preceduto da **
- Nella definizione di una funzione i parametri keyword possono essere rappresentati dall'ultimo parametro della funzione preceduto da **. In questo modo abbiamo un numero variabile di parametri keyword.
  - Il parametro è considerato un dizionario (**dict**)

45

L'operatore **

- L'operatore ** è il *mapping unpacking* operator e può essere applicato ai tipi mapping (collezione di coppie chiave-valore), quali i dizionari, per produrre una lista di coppie chiave-valore adatta ad essere passata come argomento ad una funzione.

46

Esempio

Qui **cmd** è un dizionario

```
def esempio_kw(arg1, arg2, arg3, **cmd):
    if cmd.get('operatore') == '+':
        print('La somma degli argomenti è: ', arg1 + arg2 + arg3)
    elif cmd.get('operatore') == '*':
        print('Il prodotto degli argomenti è: ', arg1 * arg2 * arg3)
    else:
        print('operatore non supportato')

    if cmd.get('azione') == "stampa":
        print('arg1 =', arg1, 'arg2 =', arg2, 'arg3 =', arg3)
```


Esempio

```
esempio_kw(2, 3, 4, operatore='+')
```
La somma degli argomenti è: 9

```
esempio_kw(2, 3, 4, operatore='*')
```
Il prodotto degli argomenti è: 24

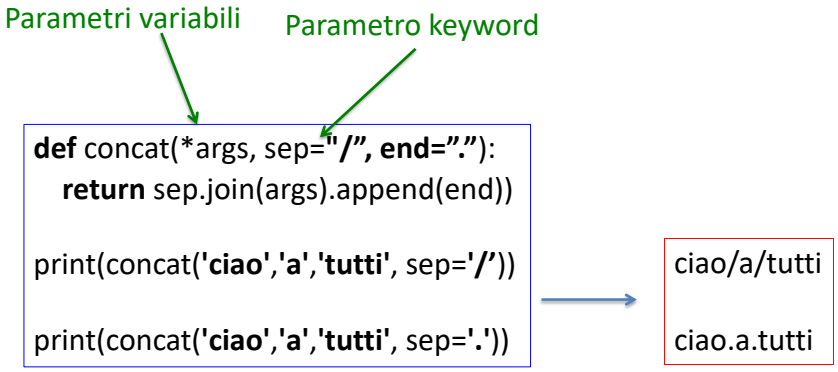
```
esempio_kw(2, 3, 4, operatore='/')
```
operatore non supportato

```
esempio_kw(2, 3, 4, operatore='+', azione='stampa')
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

```
esempio_kw(2, 3, 4, **{'operatore':'+', 'azione':'stampa'})
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

```
diz= {'operatore':'+', 'azione':'stampa'}
esempio_kw(2, 3, 4, **diz)
```
La somma degli argomenti è: 9
arg1 = 2 arg2 = 3 arg3 = 4

Esempio



49

Posizione versus Keyword

- Ci sono due modi per assegnare valori ai parametri formali di una funzione
- Secondo la posizione Parametri *tradizionali*
  Parametri *di default*
  - Gli argomenti posizionali non hanno keyword e devono essere assegnati per primi
  - La posizione è importante
- Secondo la keyword
  - Gli argomenti keyword hanno keyword e sono assegnati in seguito, dopo i parametri posizionali
  - La posizione non è importante
    - def f(x, a, b): ...
    - f('casa', a=3, b=7) è la stessa cosa di f('casa', b=7, a=3)

50

Dalla documentazione

- Ci sono 5 tipi di parametri:
- *positional-or-keyword*: specifica un argomento che puo` essere passato sia in modo posizionale o come argomento keyword. Ad esempio, foo e bar nel seguente codice:
  - **def** func(foo, bar=**None**): ...
- *positional-only*: specifica un argomento che puo` essere passato solo in modo posizionale. I parametri positional-only possono essere definiti facendoli seguire dal carattere / nella lista dei parametri della definizione della funzione. Ad esempio, *posonly1* and *posonly2* nel seguente codice sono positional-only:
  - **def** func(posonly1, posonly2, /, positional_or_keyword): ...
- *keyword-only*: specifica un argomento che puo` essere passato solo come argomento keyword. I parametri keyword-only possono essere definiti inserendo prima di loro un singolo argomento posizionale variabile o semplicemente un carattere * nella lista dei parametri. Ad esempio, *kw_only1* and *kw_only2* nel seguente codice:
  - **def** func(arg, *, kw_only1, kw_only2): ...

51

Dalla documentazione

- *var-positional*: specifica che una sequenza arbitraria di argomenti posizionali puo` essere fornita (in aggiunta agli argomenti posizionale gia` accettati da altri parametri). Questo parametro si definisce mettendo all'inizio del nome del parametro un carattere *. Ad esempio, *args* nel seguente esempio:
  - **def** func(*args, **kwargs): ...
- *var-keyword*: specifica che possono essere forniti un numero arbitrario di argomenti keyword (in aggiunta agli argomenti keyword gia` accettati da altri parametri). Questo parametro puo` essere definito attaccando ** all'inizio del nome del parametro. Ad esempio, *kwargs* nel codice in alto.

52

Riassumendo

- Una funzione può anche essere definita con tutti e tre i tipi di parametri
  - Parametri posizionali
    - Non inizializzati e di default
  - Parametro variabile
  - Parametri keyword

```
def tutti(arg1, arg2=222, *args, **kwargs):
    #Corpo della funzione
```


Esempio

```
def tutti(arg1, arg2=222, *args, **kwargs):
    print('arg1      =', arg1)
    print('arg2      =', arg2)
    print('args      =', args)
    print('kwargs =', kwargs)
```

```
tutti('prova', 999, 'uno', 2, 'tre', a=1, b='sette')
```

→

```
arg1      = prova
arg2      = 999
args      = ('uno', 2, 'tre')
kwargs = {'a': 1, 'b': 'sette'}
```

```
tutti('seconda prova')
```

→

```
arg1      = seconda prova
arg2      = 222
args      = ()
kwargs = {}
```

Annotazioni

- Le annotazioni sono dei metadati associati alle funzioni definite dal programmatore
- Sono memorizzate come un dizionario nell'attributo __annotation__ della funzione
- Non hanno nessun effetto sulla funzione
- Servono ad indicare il tipo dei parametri e del valore eventualmente restituito

55

Annotazioni

- L'annotazione di parametri è definita da **:** dopo il nome del parametro seguito da un'espressione che, una volta valutata, indica il tipo del valore dell'annotazione.
- Le annotazioni di ritorno sono definite da **->** seguita da un'espressione e sono poste tra la lista dei parametri e i due punti che si trovano alla fine dell'istruzione def.

56

Esempio

```
def saluta(nome: str, età: int = 23) -> str:
  print('Ciao ', nome, 'hai ', età, ' anni')
  return nome + ' ' + str(età)


s=saluta('mario')
print(s)
s=saluta('luisa', 21)
print(s)
```



```
Ciao  mario hai  23  anni
mario 23
Ciao  luisa hai  21  anni
luisa 21
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': <class 'str'>, 'return': <class 'str'>}
```


57

A cosa servono?

- Potrebbero essere utilizzate come help della funzione

```
def saluta(nome: 'rappresenta il nome dell\'utente ', età: int = 23) -> str:
  print('Ciao ', nome, 'hai ', età, ' anni')
  return nome + ' ' + str(età)
```

```
print(saluta.__annotations__)
```

```
{'età': <class 'int'>, 'nome': "rappresenta il nome dell'utente ", 'return': <class 'str'>}
```


58

Funzioni come parametro di funzioni

- È possibile passare l'identificatore di una funzione a come parametro di un'altra funzione b
  - Si passa il riferimento alla funzione a
- Nel corpo della funzione b, si può invocare a
  - Come nome della funzione si usa il parametro formale specificato nella definizione della funzione b

59

Esempio

```
def insertion_sort(a):
    for i in range(1,len(a)):
        val=a[i]
        j=i-1
        while (j>=0 and a[j]>val):
            a[j+1]=a[j]
            j=j-1
        a[j+1]=val
    return a
```

riferimento a funzione

```
def ordina(lista, metodo, copia=True):
    if copia == True:
        #si ordina una copia della lista
        return metodo(lista[:])
    else:
        return metodo(lista)
```

```
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = ordina(a, insertion_sort)
print('a =', a)
print('b =', b)
print('-----')
a = [5, 3, 1, 7, 8 ,2]
print('a =', a)
b = ordina(a, bubble_sort, copia=False)
print('a =', a)
print('b =', b)
```

→

```
a = [5, 3, 1, 7, 8, 2]

a = [5, 3, 1, 7, 8, 2]
b = [1, 2, 3, 5, 7, 8]
-----

a = [5, 3, 1, 7, 8, 2]

a = [1, 2, 3, 5, 7, 8]
b = [1, 2, 3, 5, 7, 8]
```


60

Espressioni **lambda**

- Funzioni anonime create usando la keyword **lambda**
- **lambda** a,b,c **:** a + b + c
  - Restituiscono la valutazione dell'espressione presente dopo i due punti
    - Può essere presente solo un'istruzione
  - Possono far riferimento a variabili presenti nello scope (ambiente) in cui sono definite
  - Possono essere restituite da funzioni
    - Una funzione che restituisce una funzione
  - Possono essere assegnate ad un identificatore
- Maggiori dettagli in seguito

61

Esempi

```
def f(x): return x**2
g = lambda x: x**2

print(g(3))
print(f(3))
```

→

```
9
9
```

f e g sono equivalenti, nel senso che producono lo stesso risultato

Ordina la lista dati considerando il valore assoluto degli elementi

```
dati = [1, -4, 2, 7, -10, -3]
print(dati)

dati.sort(key=lambda x: abs(x))
print(dati)
```

→

```
[1, -4, 2, 7, -10, -3]


[1, 2, -3, -4, 7, -10]
```


62

Funzioni Python built-in

abs()	dict()	help()	min()	setattr()
all()	dir()	hex()	next()	slice()
any()	divmod()	id()	object()	sorted()
ascii()	enumerate()	input()	oct()	staticmethod()
bin()	eval()	int()	open()	str()
bool()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	
delattr()	hash()	memoryview()	set()	

63

Output: funzione print

- Riceve un numero variabile di parametri da stampare e due parametri keyword (end e sep)
- Aggiunge automaticamente \n alla fine dell'output
- Parametri keyword (opzionali)
  - sep - stringa di separazione dell'output (default spazio)
  - end - stringa finale dell'output (default \n)
- Gli argomenti ricevuti sono convertiti in stringhe, separati da sep e seguiti da end

64

Esempi

```
dati = [1, -4, 2, 7, -10, -3]
a = 1
b = 'a'
c = 'casa'
```

```
print(a, b, c, dati)
print(a, b, c)
print(dati)
print(a, b, c, dati, sep=':')
```

→

```
1 a casa [1, -4, 2, 7, -10, -3]
1 a casa
[1, -4, 2, 7, -10, -3]
1:a:casa:[1, -4, 2, 7, -10, -3]
```

```
for v in dati:
    print(v)
```

→

```
-4
2
7
-10
-3
```

```
for v in dati:
    print(v, sep=' ')
```

→ `1 -4 2 7 -10 -3 1`

65

Output formattato

```
print('{} {}'.format('primo', 'secondo'))
print('{0} {1}'.format('primo', 'secondo'))
print('{1} {0}'.format('primo', 'secondo'))
print('{2} {0}'.format('primo', 'secondo', 'terzo'))
```

→

```
primo secondo
primo secondo
secondo primo
terzo primo
```


66

Output formattato

- Esempio di uso di format con parametri keywords

```
>>> d={"parola1":"ciao", "parola2":"?"}
>>> s="{parola1} Laura, come va {parola2}".format(**d)
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola1="ciao", parola2="?")
>>> s
'ciao Laura, come va ?'
```

```
>>> s="{parola1} Laura, come va {parola2}".format(parola2="?", parola1="ciao")
>>> s
'ciao Laura, come va ?'
```


67

Output formattato

- Consultare
  - https://docs.python.org/3/tutorial/inputoutput.html#reading-and-writing-files
- Oppure consultate il tutorial più immediato presso
  - https://pyformat.info/

68

Input: funzione input

- Riceve input da tastiera
- Può mostrare un cursore opzionale specificato come stringa
- Quello che viene letto è considerato stringa
  - Potrebbe dover essere convertito al tipo richiesto
- L'input termina con la pressione di invio (\n) che non viene inserito nella stringa letta

69

Esempi

```
a = input('Inserisci un valore: ')
print(a, type(a))
```

→

```
Inserisci un valore: e
e <class 'str'>
```

```
a = input('Inserisci un valore: ')
print(a, type(a))
```

→

```
Inserisci un valore: 12
12 <class 'str'>
```

```
a = int(input('Inserisci un valore: '))
print(a, type(a))
```

→

```
Inserisci un valore: 14
14 <class 'int'>
```


70

Lettura e scrittura di file

- La funzione built-in open() restituisce un file object che ci permette di agire sui file
- Comunemente open() è invocato con due argomenti:
  - open(filename,mode)
  - Esempio: p=open("file.txt","w")
- Il primo argomento filename è la stringa contenente il nome del file
- Il secondo argomento mode è una piccola stringa che indica in quale modalità deve essere aperto il file
  - 'r' : modalità di sola lettura
  - 'w' : modalità di sola scrittura; se il file non esiste lo crea; se il file già esiste il suo contenuto viene cancellato
  - 'a' : modalità di append; se il file non esiste lo crea; se il file già esiste il suo contenuto viene non cancellato
  - 'r+' : modalità di lettura e scrittura; il contenuto del file non viene cancellato
  - Se il secondo argomento non è specificato viene utilizzato il valore di default che è 'r'

71

Lettura e scrittura di file

Esempio: file.txt inizialmente vuoto

```
>>> fp=open("file.txt",'r+')
>>> fp.write("cominciamo a scrivere nel file")
30
>>> fp.write("\nvado al prossimo rigo")
22
```


72

Lettura e scrittura di file

- Possiamo usare close() per chiudere il file e liberare immediatamente qualsiasi risorsa di sistema usata per tenerlo aperto.
- Se il file non venisse chiuso esplicitamente, il garbage collector di Python ad un certo punto distruggerebbe il file object e chiuderebbe il file.
  - Ciò potrebbe avvenire però dopo molto tempo.
    - Dipende dall'implementazione di Python che stiamo utilizzando
- Dopo aver chiuso il file non è possible accedere in lettura o scrittura al file

73

Lettura e scrittura di file

Esempio (stesso file di prima)

```
>>> fp.close()
>>> fp.readline()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: I/O operation on closed file.
```


74

Funzioni sui file

Calling Syntax	Description
fp.read( )	Return the (remaining) contents of a readable file as a string.
fp.read(k)	Return the next k bytes of a readable file as a string.
fp.readline( )	Return (remainder of) the current line of a readable file as a string.
fp.readlines( )	Return all (remaining) lines of a readable file as a list of strings.
for line **in** fp:	Iterate all (remaining) lines of a readable file.
fp.seek(k)	Change the current position to be at the k^{th} byte of the file.
fp.tell( )	Return the current position, measured as byte-offset from the start.
fp.write(string)	Write given string at current position of the writable file.
fp.writelines(seq)	Write each of the strings of the given sequence at the current position of the writable file. This command does *not* insert any newlines, beyond those that are embedded in the strings.
print(..., file=fp)	Redirect output of print function to the file.

75

Lettura e scrittura di file

Esempio:

```
>>> f=open("newfile",'w')
>>> f.write("prima linea\n")
12
>>> f.write("seconda linea\n")
14
>>> f.write("terza linea\n")
12
>>> f.write("quarta linea\n")
13
>>> f.close()
>>> f=open('newfile','r')
>>> for line in f:
...     print(line)
...
prima linea

seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```


76

Lettura e scrittura di file

Esempio: continua dalla slide precedente

```
>>> f.seek(0)
0
>>> f.readline()
'prima linea\n'
>>> for linea in f:
...         print(linea)
...
seconda linea

terza linea

quarta linea
```

Contenuto di newfile

```
prima linea
seconda linea
terza linea
quarta linea
```


Gestione dei file

- Maggiori dettagli in
  - https://docs.python.org/3/library/filesys.html

Namespace

- Quando si utilizza un identificativo si attiva un processo chiamato risoluzione del nome (*name resolution*) per determinare il valore associato all'identificativo
- Quando si associa un valore ad un identificativo tale associazione è fatta all'interno di uno scope
- Il **namespace** (spazio dei nomi) gestisce tutti i nomi definiti in uno scope (ambito)

79

Namespace

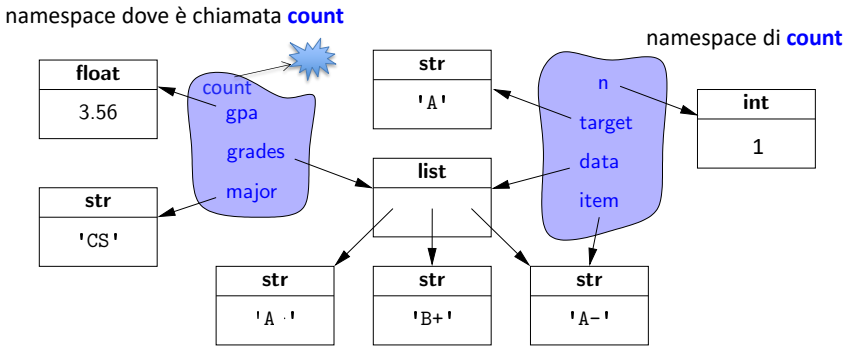
- Python implementa il namespace tramite un dizionario che mappa ogni identificativo al suo valore
- Uno scope può contenere al suo interno altri scope
- Non c'è nessuna relazione tra due identificatori che hanno lo stesso nome in due namespace differenti
- Tramite le funzioni dir() e vars() si può conoscere il contenuto del namespace dove sono invocate
  - dir elenca gli identificatori nel namespace
  - vars visualizza tutto il dizionario

80

Esempio

```
grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'
count(grades, 'A')
```

```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n
```




```
def count(data, target):
    n=0
    for item in data:
        if item == target:
            n += 1
    return n

grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'

count(grades, 'A')

print(dir())
```

Esempio

```
[
'__annotations__', '__builtins__',
'__cached__', '__doc__', '__file__',
'__loader__', '__name__', '__package__',
'__spec__', 'count', 'gpa', 'grades', 'major'
]
```

Esempio

```
def count(data, target):
    print(vars())
    n=0
    print(vars())
    for item in data:
        if item == target:
            n += 1
    print(vars())
    return n

grades = ['A', 'B+', 'A-']
gpa = 3.56
major = 'CS'

count(grades, 'A')
```

aggiungiamo print(vars())) in count

```
{'data': ['A', 'B+', 'A-'], 'target': 'A'}
{'data': ['A', 'B+', 'A-'], 'target': 'A', 'n': 0}
{'data': ['A', 'B+', 'A-'], 'target': 'A', 'n': 1, 'item': 'A-'}
```


83

I moduli in Python

- Un modulo è un particolare script Python
  - È uno script che può essere utilizzato in un altro script
  - Uno script incluso in un altro script è chiamato modulo
- Sono utili per decomporre un programma di grande dimensione in più file, oppure per riutilizzare codice scritto precedentemente
  - Le definizioni presenti in un modulo possono essere importate in uno script (o in altri moduli) attraverso il comando import
  - Il nome di un modulo è il nome del file script (esclusa l'estensione '.py')
  - All'interno di un modulo si può accedere al suo nome tramite la variabile globale __name__

84

Moduli esistenti

- Esistono vari moduli già disponibili in Python
  - Alcuni utili moduli sono i seguenti

Existing Modules	
Module Name	**Description**
array	Provides compact array storage for primitive types.
collections	Defines additional data structures and abstract base classes involving collections of objects.
copy	Defines general functions for making copies of objects.
heapq	Provides heap-based priority queue functions (see Section 9.3.7).
math	Defines common mathematical constants and functions.
os	Provides support for interactions with the operating system.
random	Provides random number generation.
re	Provides support for processing regular expressions.
sys	Provides additional level of interaction with the Python interpreter.
time	Provides support for measuring time, or delaying a program.

Utilizzare i moduli

- All'interno di un modulo/script si può accedere al nome del modulo/script tramite l'identificatore __name__
- Per utilizzare un modulo deve essere incluso tramite l'istruzione **import**
  - **import** math
- Per far riferimento ad una funzione del modulo importato bisogna far riferimento tramite il nome qualificato completamente
  - math.gcd(7,21)

Utilizzare i moduli

- Con l'istruzione from si possono importare singole funzioni a cui possiamo far riferimento direttamente con il loro nome
  - **from** math **import** sqrt
  - **from** math **import** sqrt, floor

```
import math
print(math.gcd(7,21))

from math import sqrt
print(sqrt(3))
```

→

```
7


1.7320508075688772
```

from math **import** * tutte le funzioni di math sono importate

87

Caricamento moduli

- Ogni volta che un modulo è caricato in uno script è eseguito
- Il modulo può contenere funzioni e codice *libero*
- Le funzioni sono *interpretate,* il codice libero è eseguito
- Lo script che importa (eventualmente) altri moduli ed è eseguito per primo è chiamato dall'interprete Python __main__
- Per evitare che del codice *libero* in un modulo sia eseguito quando il modulo è importato dobbiamo inserire un controllo nel modulo sul nome del modulo stesso. Se il nome del modulo è __main__ allora il codice libero è eseguito; altrimenti il codice non viene eseguito.

88

La variabile __name__

- Ogni volta che un modulo è importato, Python crea una variabile per il modulo chiamata __name__ e salva il nome del modulo in questa variabile.
- Il nome di un modulo è il nome del suo file .py senza l'estensione .py.
- Supponiamo di importare il modulo contenuto nel file test.py. La variabile __name__ per il modulo importato test ha valore "test".
- Supponiamo che il modulo test.py contenga del codice libero. Se prima di questo codice inseriamo il controllo **if** __name__ == **'__main__'**: allora il codice libero viene eseguito se e solo se __name__ ha valore __main__ . Di conseguenza, se importiamo il modulo test allora il suddetto codice libero non è eseguito.
- Ogni volta che un file .py è eseguito Python crea una variabile per il programma chiamata __name__ e pone il suo valore uguale a "__main__" . Di conseguenza se eseguiamo test.py come se fosse un programma allora il valore della sua variabile __name__ è __main__ e il codice libero dopo l'if viene eseguito.

89

Esempio

testNoIfMain.py

```
def modifica(lista):
    lista.append('nuovo')


lst = [1, 'due']
print('lista =', lst)
modifica(lst)
print('lista =', lst)
```

esecuzione testNoIfMain.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```

Stesso comportamento se eseguiti entrambi come programmi

test.py

```
def modifica(lista):
    lista.append('nuovo')

if __name__ == '__main__':
    lst = [1, 'due']
    print('lista =', lst)
    modifica(lst)
    print('lista =', lst)
```

esecuzione test.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
```


90

importUNO.py

```
import test
lista = [3,9]
print(lista)
test.modifica(lista)
print(lista)
```

esecuzione importUNO.py

```
[3, 9]
[3, 9, 'nuovo']
```

Esempio

In questo caso l'if presente in test.py evita che vengano eseguite le linee di codice libero presenti in test.py

importDUE.py

```
import testNoIfMain
lista = [3,9]
print(lista)
testNoMain.modifica(lista)
print(lista)
```

esecuzione importDUE.py

```
lista = [1, 'due']
lista = [1, 'due', 'nuovo']
[3, 9]
[3, 9, 'nuovo']
```

In questo caso vengono eseguite anche le linee di codice libero di testNoIfMain.py perché non sono precedute dall'if

91

package

- Modo per strutturare codice Python in moduli, cartelle e sotto-cartelle
- Il package è una collezione di moduli
  - Il package è una cartella in cui, oltre ai moduli o subpackage, è presente il file __init__.py che contiene istruzioni di inizializzazione del package (può essere anche vuoto)
  - __init__.py serve ad indicare a Python di trattare la cartella come un package

92

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

In uno script presente nella cartella che contiene **sound**

```
import sound.effects.echo
sound.effects.echo.echofilter(input, output, delay=0.7)

from sound.effects import echo
echo.echofilter(input, output, delay=0.7)

from sound.effects.echo import echofilter
echofilter(input, output, delay=0.7)
```


93

```
sound/                          Top-level package
      __init__.py               Initialize the sound package
      formats/                  Subpackage for file format conversions
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
      effects/
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
      filters/
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

Per importare moduli in surround.py si usa un import relativo

```
from . import echo
from .. import formats
from ..filters import equalizer
```

N.B. gli import relativi si basano sul nome del modulo corrente. Siccome il nome del modulo main e` sempre "__main__", i moduli usati come moduli main devono sempre usare import assoluti.

94

Importare moduli tra package

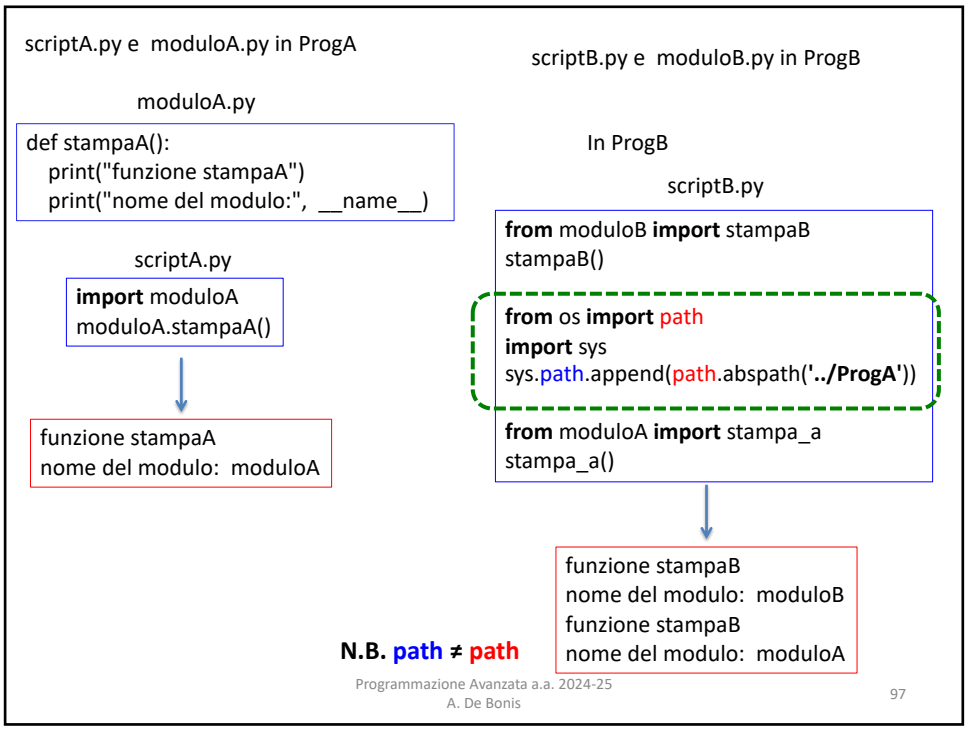
- Lo script che importa il modulo deve conoscere la posizione del modulo da importare
  - Non è necessario quando
    - il modulo è un modulo di Python
    - il modulo è stato installato
  - La variabile sys.path è una lista di stringhe che determina il percorso di ricerca dell'interprete Python per i moduli
  - Occorre aggiungere a sys.path il percorso assoluto che contiene il modulo da importare

95

Importare moduli tra package

- Quando il modulo miomodulo è importato l'interprete prima cerca un modulo built-in con quel nome. Se non lo trova, cerca un file miomodulo.py nella lista di directory date dalla variabile **sys.path**
- sys.path e` una lista di stringhe che specifica il percorso di ricerca di un modulo e contiene nella prima posizione la directory contenente lo script input
- sys.path è inizializzata dalle seguenti locazioni:
  - e` inizializzata da PYTHONPATH (una lista di nomi di directory con la stessa sintassi della variabile shell PATH).
  - Default dipendente dall'installazione

96



97

Python e OOP

- Python supporta tutte le caratteristiche standard della OOP
  - Derivazione multipla
  - Una classe derivata può sovrascrivere qualsiasi metodo della classe base
- Tutti i membri di una classe (dati e metodi) **sono pubblici**

98

Ereditarietà

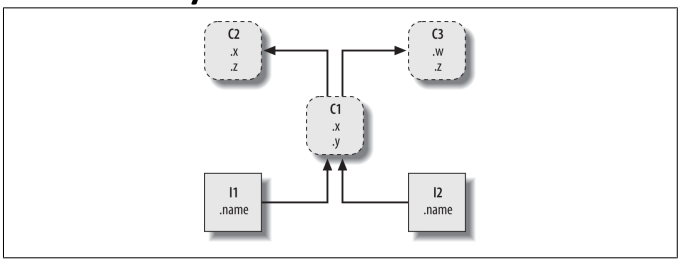
- Le superclassi di una classe vengono elencate tra le parentesi nell'intestazione della classe
- Le superclassi potrebbero trovarsi in altri moduli
  - Esempio: supponiamo che FirstClass sia nel modulo modulename

```
from modulename import FirstClass
class SecondClass(FirstClass):
    def display(self): ...
```

oppure

```
import modulename
class SecondClass(modulename.FirstClass):
    def display(self): ...
```


Python e OOP



- I1.w viene risolto in C3.w
- Python cerca l'attributo nell'oggetto e poi risale man mano nelle classi sopra di esso dal basso verso l'alto e da sinistra verso destra
  - I2.z viene risolto in C2.z

Classi in Python

- In Python in una classe possiamo avere
  - variabili di istanza (dette anche membri dati)
  - variabili di classe
    - condivise tra tutte le istanze della classe
  - metodi (detti anche membri funzione)
    - metodi specifici della classe
    - overloading di operatori
- Per far riferimento ad una variabile di istanza si fa precedere l'identificatore dalla parola chiave **self**
  - se non esiste una variabile di istanza con lo stesso nome, **self** puo` essere usato anche per far riferimento ad una variabile di classe

101

Attributi di classe e attributi di istanza

- Le variabili di classe sono di solito (ma non solo) aggiunte alla classe mediante assegnamenti all'esterno delle funzioni.
- Le variabili di istanza possono essere aggiunte all'istanza mediante assegnamenti effettuati all'interno di funzioni che hanno self tra gli argomenti.

102

Attributi di classe e attributi di istanza

```
class myClass:
      a=3
      def method(self):
            self.a=4


x=myClass()
print(x.a)
x.method()
print(x.a)
y=myClass()
print(y.a)
print(myClass.a)

x.b=10
print(x.b)
```

```
3
4
3
3
10
```