Programmazione avanzata a.a. 2019-20
A. De Bonis

Introduzione a Python (II parte)

shallow vs deep copy

- Dal manuale Python
  - A shallow copy constructs a new compound object and then inserts references into it to the objects found in the original
  - *Costruisce un nuovo oggetto composto e inserisce in esso i riferimenti agli oggetto presenti nell'originale*
  - A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original
  - *Costruisce un nuovo oggetto composto e ricorsivamente inserisce in esso le copie degli oggetti presenti nell'orginale*

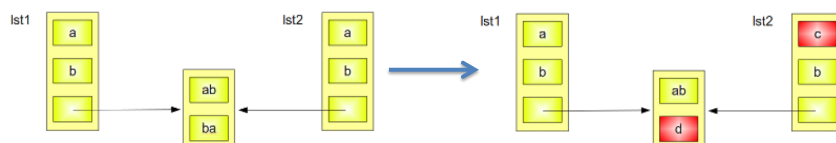
Problemi con deep copy

- Dal manuale Python
  1. Recursive objects (compound objects that, directly or indirectly, contain a reference to themselves) may cause a recursive loop
     - Ad esempio, se un oggetto **a** contiene un riferimento a se stesso allora una copia deep di **a** causa un loop
  2. Because deep copy copies everything it may copy too much, e.g., even administrative data structures that should be shared even between copies

Esempio shallow/deep copy

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy()  #metodo della classe list
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

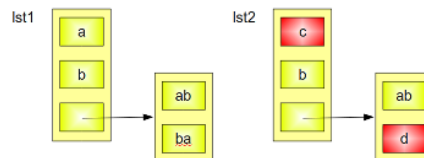
```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'd']]
lista2 = ['c', 'b', ['ab', 'd']]
```



Esempio shallow/deep copy

```
from copy import deepcopy
lst1 = ['a','b',['ab','ba']]
lst2 = deepcopy(lst1)
print('lista1 =', lst1)
print('lista2 =', lst2)
lst2[0] = 'c'
lst2[2][1] = 'd'
print('lista1 =', lst1)
print('lista2 =', lst2)
```

```
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['a', 'b', ['ab', 'ba']]
lista1 = ['a', 'b', ['ab', 'ba']]
lista2 = ['c', 'b', ['ab', 'd']]
```



Espressioni ed operatori

- Espressioni esistenti possono essere combinate con simboli speciali o parole chiave (operatori)
- La semantica dell'operatore dipende dal tipo dei suoi operandi

```
a=3
b=4
c=a+b
print('a+b =', c)
a='ciao '
b='mondo'
c=a+b
print('a+b =', c)
```

```
a+b = 7
a+b = ciao mondo
```

Operatori aritmetici

- Gli operatori aritmetici sono quelli a destra

+	addition
−	subtraction
*	multiplication
/	true division
//	integer division
%	the modulo operator

- Per gli operatori +, -, *
  - Se entrambi gli operandi sono int, il risultato è int
  - Se uno degli operandi è float, il risultato è float
- Per la divisione *vera* /
  - Il risultato è sempre float
- Per la divisione intera //
  - Il risultato (int) è la parte intera della divisione

// e % definiti anche per numeratore o denominatore negativo. Dettagli sul manuale

// adesso si chiama floor division

Operatori logici
Operatori di uguaglianza

- Python supporta i seguenti operatori logici

not	unary negation
and	conditional and
or	conditional or

- Python supporta i seguenti operatori di uguaglianza

is	same identity
is not	different identity
==	equivalent
!=	not equivalent

Operatori di uguaglianza

- L'espressione a **is** b risulta vera solo se a e b sono alias dello stesso oggetto
- L'espressione a **==** b risulta vera anche quando gli identificatori a e b si riferisco ad oggetti che possono essere considerati equivalenti
  - Due oggetti dello stesso tipo che *contengono* gli stessi valori

Esempio

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1
if lst1 is lst2:
   print('Oggetti identici')
else:
   print('Oggetti distinti')

if lst1 == lst2:
   print('Oggetti equivalenti')
else:
   print('Oggetti non equivalenti')
```

Oggetti identici
Oggetti equivalenti

```
lst1 = ['a','b',['ab','ba']]
lst2 = lst1.copy()
if lst1 is lst2:
   print('Oggetti identici')
else:
   print('Oggetti distinti')

if lst1 == lst2:
   print('Oggetti equivalenti')
else:
   print('Oggetti non equivalenti')
```

Oggetti distinti
Oggetti equivalenti

Operatori di confronto

- Python supporta i seguenti operatori di confronto

<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to

- Per gli interi hanno il significato atteso
- Per le stringhe sono case-sensitive e considerano l'ordinamento lessicografico
- Per sequenze ed insiemi assumono un significato particolare (dettagli in seguito)

Operatori per sequenze

list, tuple e str

- I tipi sequenza predefiniti in Python supportano i seguenti operatori

s[j]	element at index j
s[start:stop]	slice including indices [start,stop)
s[start:stop:step]	slice including indices start, start + step, start + 2∗step, . . . , up to but not equalling or stop
s + t	concatenation of sequences
k ∗ s	shorthand for s + s + s + ... (k times)
val **in** s	containment check
val **not in** s	non-containment check

```
t = [2] * 7
print(t)
```

→ [2, 2, 2, 2, 2, 2, 2] ←

```
t = 7 * [2]
print(t)
```

Indici negativi

- Le sequenze supportano anche indici negativi
- s[-1] si riferisce all’ultimo elemento di s
- s[-2] si riferisce al penultimo elemento di s
- s[-3] …

- s[j] = val sostituisce il valore in posizione j
- **del** s[j] rimuove l’elemento in posizione j

Confronto di sequenze

- Le sequenze possono essere confrontate in base all’ordine lessicografico
  - Il confronto è fatto elemento per elemento
  - Ad esempio, [5, 6, 9] < [5, 7] (True)

s == t	equivalent (element by element)
s != t	not equivalent
s < t	lexicographically less than
s <= t	lexicographically less than or equal to
s > t	lexicographically greater than
s >= t	lexicographically greater than or equal to

Operatori per insiemi

- Le classi **set** e **frozenset** supportano i seguenti operatori

key **in** s	containment check
key **not in** s	non-containment check
s1 == s2	s1 is equivalent to s2
s1 != s2	s1 is not equivalent to s2
s1 <= s2	s1 is subset of s2
s1 < s2	s1 is proper subset of s2
s1 >= s2	s1 is superset of s2
s1 > s2	s1 is proper superset of s2
s1 | s2	the union of s1 and s2
s1 & s2	the intersection of s1 and s2
s1 − s2	the set of elements in s1 but not s2
s1 ˆ s2	the set of elements in precisely one of s1 or s2

Operatori per dizionari

- La classe **dict** supporta i seguenti operatori

d[key]	value associated with given key
d[key] = value	set (or reset) the value associated with given key
del d[key]	remove key and its associated value from dictionary
key **in** d	containment check
key **not in** d	non-containment check
d1 == d2	d1 is equivalent to d2
d1 != d2	d1 is not equivalent to d2

Precedenza degli operatori

priorità

	Operator Precedence	
	Type	**Symbols**
1	member access	expr.member
2	function/method calls container subscripts/slices	expr(...) expr[...]
3	exponentiation	**
4	unary operators	+expr, −expr, ˜expr
5	multiplication, division	*, /, //, %
6	addition, subtraction	+, −
7	bitwise shifting	<<, >>
8	bitwise-and	&
9	bitwise-xor	^
10	bitwise-or	|
11	comparisons containment	**is**, **is not**, ==, !=, <, <=, >, >= **in**, **not in**
12	logical-not	**not** expr
13	logical-and	**and**
14	logical-or	**or**
15	conditional	val1 **if** cond **else** val2
16	assignments	=, +=, −=, *=, etc.

Assegnamento esteso

- In C o Java gli operatori binari ammettono una versione *contratta*
  - i += 3 è equivalente a i = i + 3
- Tale caratteristica esiste anche in Python
  - Per i tipi immutable si crea un nuovo oggetto a cui si assegna un nuovo valore e l'identificatore è riassegnato al nuovo oggetto
  - Alcuni tipi di dato (e.g., list) ridefiniscono la semantica dell'operatore +=

Chaining

- Assegnamento
  - In Python è permesso l'assegnamento concatenato
  - x = y = z = 0
- Operatori di confronto
  - In Python è permesso 1 < x + y <= 9
  - Equivalente a (1 < x+y) **and** (x + y <= 9), ma l'espressione x+y è calcolata una sola volta

```
x=y=5
if 3 < x+y <= 10:
   print('interno')
else:
   print('esterno')
```


Esempio += per list

```
alpha = [1, 2, 3]
beta = alpha
print('alpha =', alpha)
print('beta  =', beta)
beta += [4, 5]
print('beta  =', beta)
beta = beta + [6, 7]
print('beta  =', beta)
print('alpha =', alpha)
```

```
alpha = [1, 2, 3]
beta  = [1, 2, 3]
beta  = [1, 2, 3, 4, 5]
beta  = [1, 2, 3, 4, 5, 6, 7]
alpha = [1, 2, 3, 4, 5]
```

beta += [4, 5] estende la lista originale

Equivalente a beta.extend([4,5])

beta = beta + [6, 7] riassegna beta ad una nuova lista

Riferimenti

- M. Summerfield, “Programming in Python 3. A Complete Introduction to the Python Language” , Addison-Wesley
- M. Lutz, "Learning Python», 5th Edition,O'Reilly
- Altro materiale sarà indicato in seguito

Controllo del flusso in Python

Blocchi di codice

- In Python i blocchi di codice non sono racchiusi tra parentesi graffe come in C o Java
- In Python per definire i blocchi di codice o il contenuto dei cicli si utilizza l'indentazione
  - Ciò migliora la leggibilità del codice, ma all'inizio può confondere il programmatore

Indentazione del codice: Spazi o tab

- Il metodo preferito è indentare utilizzando spazi (di norma 4)
- Il tab può essere diverso tra editor differenti
- In Python 3 non si possono mischiare nello stesso blocco spazi e tab
  - In Python 2 era permesso

Stile per Codice Python
https://www.python.org/dev/peps/pep-0008/#tabs-or-spaces

if elif … else

```
if first_condition:
    first_body
elif second_condition:
    second_body
elif third_condition:
    third_body
else:
    fourth_body
```

codice indentato

Se il blocco è costituito da una sola istruzione, allora può andare subito dopo i due punti

elif ed **else** sono opzionali

```
if x < y and x < z:
    print('x è il minimo')
elif y < z:
    print('y è il minimo')
else:
    print('z è il minimo')
```


Esempi

```
print('inizio')
if 5>3:
    print(1)
    print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```

```
inizio
1
2
3
4
fine
```

```
print('inizio')
if 5>3:
    print(1)
 print(2)
    print(3)
    print(4)
else:
    print(5)
    print(6)
    print(7)
print('fine')
```

ERRORE

while

while *condition*:
    body

```
j=0
while j < len(data) and data[j] != X:
    j += 1
```

```
while a<b:
    print(a)
    a=a+1
```


for … in

for *element* **in** *iterable*:
    body   # body may refer to 'element' as an identifier

```
total = 0
for val in data:
    total += val
```

```
biggest = 0
for val in data:
    if val > biggest:
        biggest = val
```

range()

- range(n) genera una lista di interi compresi tra 0 ed n-1
  - range(start, stop, step)

```
>>> list(range(1,10,3))
[1, 4, 7]
```

- Utile quando vogliamo iterare in una sequenza di dati utilizzando un indice
  - **for** i **in range**(n)

```
for i in range(0, -10, -2): print(i)
```

```
0
-2
-4
-6
-8
```

```
big_index = 0
for j in range(len(data)):
  if data[j] > data[big_index]:
    big_index = j
```


Esempi

```
for _ in range(1,6):
  print('ciao')
```

```
ciao
ciao
ciao
ciao
ciao
```

```
# Stampa la lunghezza delle
# parole in una lista
words = ['cat', 'window', 'defenestrate']
for w in words:
  print(w, len(w))
```

```
cat 3
window 6
defenestrate 12
```

```
# Cicla su una copia della lista
for w in words[:]:
  if len(w) > 6:
    words.insert(0, w)
print(words)
```

```
['defenestrate', 'cat', 'window', 'defenestrate']
```

```
# Cicla su sulla stessa lista
for w in words:
  if len(w) > 6:
    words.insert(0, w)
print(words)
```

Crea una lista infinita

break e **continue**

- **break** termina immediatamente un ciclo for o while, l'esecuzione continua dall'istruzione successiva al while/for
- **continue** interrompe la corrente iterazione di un ciclo for o while e continua verificando la condizione del ciclo

```
found = False
for item in data:
  if item == target:
    found = True
    break
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

79

Clausola **else** e cicli

- Utilizzata con cicli che prevedono un **break**
- La clausola **else** è eseguita quando si esce dal ciclo ma non a causa del **break**

```
n=3
for x in [4, 5, 7, 8, 10]:
    if x % n == 0:
        print(x, ' è un multiplo di ', n)
        break
else:
   print('non ci sono multipli di', n , 'nella lista')
```

```
non ci sono multipli di 3 nella lista
```

Con n=2

```
4  è un multiplo di  2
```

Programmazione Avanzata a.a. 2019-20
A. De Bonis

80

Python: if *abbreviato*

- In C/Java/C++ esiste la forma abbreviata dell'if

massimo = a > b ? a : b

- Anche Python supporta questa forma, ma la sintassi è differente

massimo = a if (a > b) else b

List Comprehension

- *Comprensione di lista*
- Costrutto sintattico di Python che agevola il programmatore nella creazione di una lista a partire dall'elaborazione di un'altra lista
  - Si possono generare tramite comprehension anche
    - Insiemi
    - Dizionari

[expression **for** value **in** iterable **if** condition]

List Comprehension

- *expression* e *condition* possono dipendere da *value*
- La parte **if** è opzionale
  - In sua assenza, si considerano tutti i *value* in iterable
  - Se *condition* è vera, il risultato di *expression* è aggiunto alla lista
- **[** *expression* **for** *value* **in** iterable **if** *condition* **]** è equivalente a

```
result = [ ]
for value in iterable:
   if condition:
      result.append(expression)
```


Esempi

Lista dei quadrati dei numeri compresi tra 1 ed n

```
squares = [k*k for k in range(1, n+1)]
```

Lista dei divisori del numero n

```
factors = [k for k in range(1,n+1) if n % k == 0]
```

```
[str(round(pi, i)) for i in range(1, 6)]
```

→

```
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Doppia comprehension

```
[(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
```

```
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

```
combs = []
for x in [1,2,3]:
  for y in [3,1,4]:
    if x != y:
      combs.append((x, y))
```

```
a = [(x, y) for x in [1,2,3] for y in ['a', 'b', 'c']]
print(a)
```

```
[(1, 'a'), (1, 'b'), (1, 'c'), (2, 'a'), (2, 'b'), (2, 'c'), (3, 'a'), (3, 'b'), (3, 'c')]
```


Doppia comprehension

```
matrix = [ [1, 2, 3, 4],  [5, 6, 7, 8], [9, 10, 11, 12] ]
print(matrix)
transposed = [[row[i] for row in matrix] for i in range(4)]
print(trasposed)
```

```
[ [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12]]
```

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Altri tipi di comprehension

- list comprehension

  [k*k for k in range(1, n+1)]

- set comprehension

  { k*k for k in range(1, n+1) }

- dictionary comprehension

  { k : k*k for k in range(1, n+1) }