POLITECNICO DI MILANO
Piazza Leonardo da Vinci, 32 - 20133 Milano
Tel. +39.02.2399.1 - http://www.polimi.it

Introduzione a Python

Matteo Pradella
Paolo Costa
Matteo Migliavacca

Python sta per Pitone?

- No. Il nome deriva da “Monty Python's Flying Circus” (gruppo di comici inglese)
- Show BBC ma anche film: ricordiamo tra gli altri Brian di Nazareth, Il Senso della Vita, E ora qualcosa di completamente diverso…

- Guido van Rossum (padre di Python) e` un fan…

Dove trovare informazioni...

- Sito ufficiale del linguaggio: interprete linux / win / MacOS (ultima versione 2.3), IDE per Win, tutorial, reference,...
  http://www.python.org
- “Dive into Python” (free book molto completo)
  http://diveintopython.org/index.html
- "How to Think Like a Computer Scientist with Python“ (più introduttivo)
  http://greenteapress.com/thinkpython
- Google -> Python

Un linguaggio interpretato

- Python, a differenza di C/C++, e` interpretato (anche se poi molte implementazioni lo compilano per motivi di efficienza):
- si puo` interagire con una macchina virtuale Python in maniera interattiva
- Prompt: >>>
- # questo e` un commento (come // in C++)

Es.

```
>>> 2+2
4
>>> 10 / 3      # divisione intera
3
```

Definizione variabili

```
>>> pippo = 7
```

Abbiamo creato una variabile pippo, di tipo intero, contenente il valore 7

```
>>> pippo = 5.5
```

Pippo e` divenuta una variabile reale…

```
>>> a = b = c = 0
```

sia a che b che c assumono il valore 0

Stringhe

Le stringhe si possono scrivere in vari modi:

```
'questi sono'
'un po\' di caratteri'
```

, oppure, se preferisco:

```
"un po' di caratteri"
'Mi guardo` e mi disse: "vattene!" Allora gli sparai...'
```

`'\n'` *va a capo come in C*

Stringhe:concatenazione, ripetizione

```
>>> ehm = "aiuto! "
>>> “Al fuoco! ” + ehm
Al fuoco! aiuto!
>>> ehm * 7
'aiuto! aiuto! aiuto! aiuto! aiuto! aiuto! aiuto! '
>>> ehm * 0
' ' # stringa vuota
```

Stringhe e indici

In Python le stringhe sono sequenze: si puo` accedere ad elementi tramite *indici*

```
>>> "Questa lezione mi sta annoiando parecchio"[4]
't'
>>> "Preferivo stare a letto"[0:9]    # ecco uno slice
'Preferivo'
```

N.B. 9 escluso nello *slice*

Non cambiar la stringa

Le stringhe non sono modificabili

```
>>> casa = "voglio andare a casa"; casa[4] = 'i'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

Pero` posso copiarle:

```
>>> casetta = casa[0:5] + 'a' + casa[6:]
```

Ancora slice

```
>>> parola = "casa"
>>> parola[:2]
'ca'
>>> parola[2:]
'sa'
>>> parola[:]
'casa' # N.B. e` una copia
>>> parola[-2] # anche all'indietro!
's'
>>> len(parola) # lunghezza di una sequenza
4
```

c	a	s	a	
0	1	2	3	4
-4	-3	-2	-1	

Liste

- Fondamentali e molto usate in Python
- Sono sequenze, come le stringhe (ergo: indici, slice, len)
- Pero` sono modificabili
- Assumono anche il ruolo che e` degli *array* in altri linguaggi

es.

```
>>> lista = ["una lista", 4, 6.2]
>>> len(lista)
3
```

append

un classico delle liste: si aggiunge elemento in coda con *append*

```
>>> lista.append(3)
>>> lista
['una lista', 4, 6, 2, 3]
```

E ora i cicli…

Partiamo col ciclo while

un semplice esempio: i numeri di Fibonacci

```
a, b = 0, 1    # assegnamento con tupla (detto multiplo)!
while b < 10:
   print b, # "," serve ad evitare \n finale
   a, b = b, a+b
```

come delimito il blocco di istruzioni?

(in C/C++ si usa {…})

Stile di indentazione

- In Python, a differenza della stragrande maggioranza degli altri linguaggi, il corpo del while (un *blocco* in generale) e` delimitato per mezzo della *indentazione*!

```
→ istr1
    istr2
    → istr3
    → istr4
    → istr5
```

blocco 2

blocco 1

```
{ istr1;  // in C
  istr2;
  {
     istr3;
     istr4;
     istr5;
  }}
```

(posso usare spazi o tab, basta che siano lo stesso numero)

if

- La forma piu` generale:

```
if cond1 :
   # cond1 vera
elif cond2 :
   # cond2 vera
elif cond3 :
   # cond3 vera
...
else :
   # nemmeno una vera!
```

Ora il *for*

- Il for, a differenza del C, itera su sequenze (es. stringhe o liste)
- in pratica:

```
for i in seq:
 # fai qualcosa con i
```

Per esempio:

```
>>> for i in ['Ma questa', 'e`', 1, 'lista?'] :
...     print i,
Ma questa e` 1 lista?
```

range()

- range permette di iterare su una sequenza di numeri (senza doverli scrivere tutti)

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Come al solito, estremo superiore escluso.

Costruttori sofisticati di liste

- Comodi! Assomigliano molto alla notazione insiemistica, per es. un analogo di {(x,y) | x ∈ A, y ∈ B, x ≠ y} si puo` scrivere come

  ```
  [(x,y) for x in A for y in B if x != y]
  ```

- A proposito, (x,y) e` un esempio di tupla (come in matematica) – in Python e` una *sequenza immutabile*
  - si puo` usare come alternativa piu` efficiente alle liste, se non si devono modificare dati
- Attenzione! E` comunque una *lista*, non un insieme (e` un insieme totalmente ordinato che ammette piu` occorrenze dello stesso elemento)…

while

- Oltre al ciclo for, eiste anche il ciclo while

```
while(condizione):
    istruz1
    istruz2
    istruz3
    ...
```

Le funzioni

- Possiamo definire funzioni con *def*:

```
def f(n) :
   """Be' se proprio voglio qui ci metto la stringa di
   documentazione (cosa fa f?)"""
   if n == 0:
    return 1
   else:
    return n*f(n-1)
```

Scope (o campo d'azione) statico

- Capisco dalla *struttura statica* del programma dove sono definiti i nomi ad es. di variabile che sto usando

```
>>> def f():
    x = 5
    g()
>>> def g():
    print x

>>> f()

Traceback (most recent call last):
  File "<pyshell#7>", line 1, in -toplevel-
    g()
  File "<pyshell#6>", line 2, in g
    print x
NameError: global name 'x' is not defined
```

Funzioni: argomenti di default

```
def incipitizza(seq, incipit = 'banale'):
   seq[0] = incipit
   return seq

>>> a = [1,2,3]
>>> incipitizza(a)
['banale', 2, 3]
>>> incipitizza(a,1)
...
```

Argomenti con parole chiave

```
def parrot(voltage, state='a stiff', action='voom',
   type='Norwegian Blue'):
   print "-- This parrot wouldn't", action,
   print "if you put", voltage, "Volts through it."
   print "-- Lovely plumage, the", type
   print "-- It's", state, "!"
```

Si puo` chiamare in questi modi:

```
parrot(1000)
parrot(action = 'VOOOOOM', voltage = 1000000)
parrot('a thousand', state = 'pushing up the daisies')
parrot('a million', 'bereft of life', 'jump')
```

- In questo modo nell'invocazione della funzione posso alternare indifferentemente parametri attuali e parametri di default (purchè il risutlato finale sia consistente)

I dizionari

- Sono anche chiamati *memorie associative* o *array associativi*
- A differenza delle sequenze, gli indici non sono interi bensi`*chiavi* (es. stringhe)
- Sintassi: `{chiave1 : val1, chiave2 : val2, ...}`
- il metodo `keys()` restituisce la lista delle chiavi di un dizionario

dizionari: qualche esempio…

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'sape': 4139, 'guido': 4127, 'jack':
4098}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'guido': 4127, 'irv': 4127, 'jack':
4098}
>>> tel.keys()
['guido', 'irv', 'jack']
>>> tel.has_key('guido')
True
```

dir()

- `dir()` applicato a qualcosa mi dice quali nomi sono definiti in questo qualcosa (un po' vago…)
- esempio:

```
>>> a = [1, 2, 3]
>>> dir(a)['__add__', '__class__', '__contains__',
   '__delattr__', '__delitem__', '__delslice__', '__doc__',
   '__eq__', '__ge__', '__getattribute__', '__getitem__',
   '__getslice__', '__gt__', '__hash__', '__iadd__',
   '__imul__', '__init__', '__iter__', '__le__', '__len__',
   '__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
   '__reduce_ex__', '__repr__', '__rmul__', '__setattr__',
   '__setitem__', '__setslice__', '__str__', 'append',
   'count', 'extend', 'index', 'insert', 'pop', 'remove',
   'reverse', 'sort']
```

NB: la notazione *__qualcosa__* (dove __ sono due caratteri di sottolineatura) e` abbastanza classica in Python: sono metodi ed attributi con ruoli particolari – vedremo meglio nella parte sulla OO

Classi e programmazione OO

- Python da 2.2 introduce le cosiddette nuove *classi* noi vedremo esclusivamente queste -- le altre rimangono per ragioni di compatibilità
- NB: non guardate il tutorial (anche ultima versione) perchè non è aggiornato. Consultate il *What's new in 2.2*

Definizione di classe

```
class NomeClasse(object) :
   a = 5
   def __init__(self, altro) :
    ...
   ...
```

il progenitore di ogni classe è ***object***

self è *sempre* il primo argomento di un metodo: si riferisce all'oggetto stesso (self, appunto!)

costruttore

```
a = “foofoo”
x = NomeClasse(a)
```

attributi sono modificabili dinamicamente e accessibili con la notazione puntata (es. oggetto.attributo)

elementi "privati"

- Tutto e` pubblico, in genere (invece ad es. in C++ devo definire la parte visibile per mezzo di *public:*)
- un modo per "nascondere" metodi e attributi, e` dare loro un nome che inizia con un doppio '_' – per es. *__pippo*
- l'attributo/metodo creato in questo caso ha il nome effettivo *_nomedellaclasse__pippo*, piuttosto difficile da usare per errore...

Ereditarieta`

```
>>> class A(object):
        def f(self):
            print "Padre"
>>> class B(A):
        def f(self):
            print "Figlio"
>>> x = A()
>>> y = B()
>>> x.f()
Padre
>>> y.f()
Figlio
```

- Supporta l'ereditarieta' multipla

Overload

- Non posso fare overloading (se ho piu' metodi con numero diverso di parametri viene chiamato quello che ho definito per ultimo)

```
>>> class A(object):
def f(self):
 print “metodo senza parametri"
def f(self, n):
 print “metodo con parametri"
>>> x = A()
>>> x.f()

Traceback (most recent call last):
File "<pyshell#7>", line 1, in -toplevel-
 x.f()
TypeError: f() takes exactly 2 arguments (1 given)
>>> x.f(2)
```

Python Type System

- E’ un linguaggio tipizzato
- Non fa static type checking (ma fa dynamic type checking: gli errori di tipo vengono rilevati, ma a run-time)
- Il tipo delle variabili non e’ dichiarato

```
>>> x = 0 # x bound to an integer object
>>> x = "Hello" # now it's a string
>>> x = [1, 2, 3] # and now it's a list
```

Python Type System

- Le funzioni sono first class object
- Non c'e` nessuna differenza tra variabili che contengono valori e quelle che contengono funzioni (callable/non callable)

```
>>> def f():
>>>         return 66
>>> x = f
>>> x()
66
```

- Posso anche passare le funzioni come parametri

```
>>> def f(x):
   x()
>>> def a():
   print "Io sono a"
>>> def b():
   print "Io sono b"
>>> f(a)
Io sono a
>>> f(b)
Io sono b
```

Dynamic Type Checking

- Ottengo un errore di tipo quando python non trova l'attributo a cui sto accedendo cioe':
  - invoco un metodo non definito dell'oggetto
  - leggo un campo non definito dell'oggetto

Dynamic Type Checking

```
class libro(object):
    def __init__(self,contenuto):
        self.contenuto = contenuto
    contenuto = "Nel mezzo del cammin di nostra vita"
    def read(self):
        return self.contenuto

def stampaContenuto(l):
    print l.read()

x = libro("Nel mezzo del cammin di nostra vita")

y = "Questo non e' un libro"
import random
if random.random() < 0.5:
    stampaContenuto(y)
else:
   stampaContenuto(x)
```

Questo programma ha un errore di tipo ma viene rilevato nel 50% dei casi

Dynamic Type Checking

- In questo caso ottengo un eccezione

```
Traceback (most recent call last):
  File "<pyshell#215>", line 2, in -toplevel-
    stampaContenuto(y)
  File "<pyshell#206>", line 2, in stampaContenuto
    print l.read()
AttributeError: 'str' object has no attribute 'read'
```

Dynamic Type Checking

- MA:

```
class P(object):
    valore = 5
>>> x = P()
>>> x.valore = 10
>>> x.valore
10
>>> x.valoree = 20
```

- Nessuna eccezione!! => in python se provo ad assegnare (binding) un attributo che non esiste python lo crea al momento!

Dynamic Type Checking

- Quindi in questo momento la variabile x e una reference a un istanza di P con IN PIU' un attributo "valoree"

```
>>> type(x)
<class '__main__.P'>
>>> dir(x)
['__class__', '__delattr__', '__dict__', '__doc__',
   '__getattribute__', '__hash__', '__init__', '__module__',
   '__new__', '__reduce__', '__reduce_ex__', '__repr__',
   '__setattr__', '__str__', '__weakref__', 'valore',
   'valoree']
```

Dynamic Type Checking

- Risoluzione degli attributi (es x.attr)
  - prima cerco nell'istanza x
  - poi cerco nella classe di x
  - infine cerco nelle classi padre (ereditarieta')

Tutto dinamico…

- Oltre le istanze posso modificare dinamicamente anche la classe
- Tutte le istanze da quel momento in poi hanno i nuovi attributi (ovvio se pensiamo alla procedura di risoluzione degli attributi)

```
class p(object) :
    def
__init__(self) :
        self.a = 5

def f(t) : # da
aggiungere
    print t.a

>>> x = p()
>>> p.f = f # lo
aggiungiamo
>>> x.f()
5
```

```
def g(t,n) :
    t.a = n


>>> p.g = g
>>> x.g(3)
>>>x.f()
3
```

Assegnamento - 1

- In Python l'accesso agli oggetti avviene tramite reference (analogo di quanto avviene in JAVA con le classi)
- Non esiste dichiarazione delle varibili: vengono istanziate quando vi si assegna un valore per la prima volta
- Non si può utilizzare una variabile prima che sia stata inizializzata

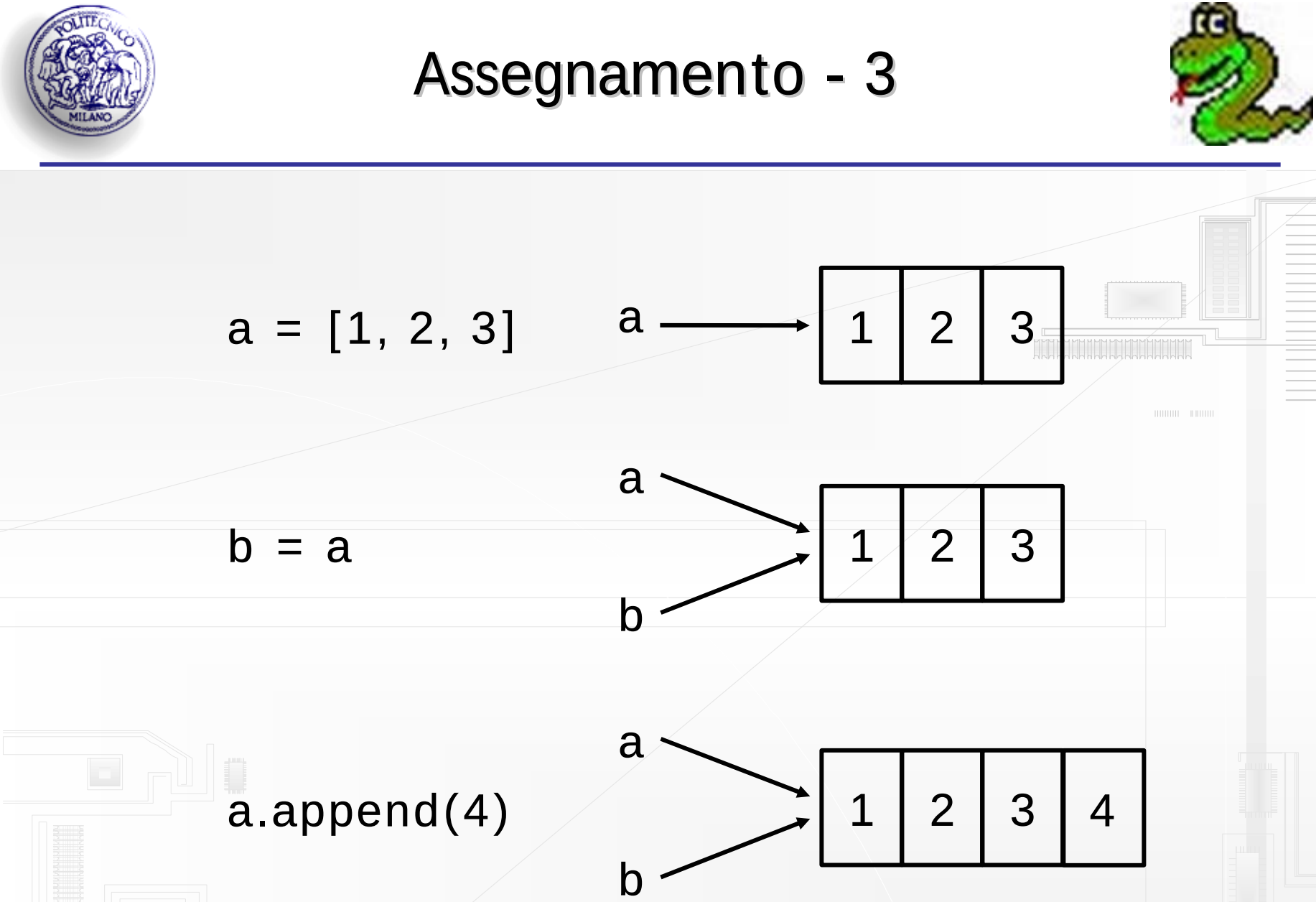
Assegnamento - 2

- Quando si esegue un assegnamento in realtà viene copiata la reference non l'oggetto

```
>>> a = [1,2,3]
>>> b = a
>>> id(a) # id(var) restituisce l'indirizzo (l-value) di var
135533752
>>> id(b)
135533752
```

- Si crea un alias: modificando a modifico anche b

Assegnamento - 3



Assegnamento - 4

In Python gli oggetti si dividono in:

- Oggetti mutabili il cui valore può essere modificato (liste, dizionari, classi)

```
>>> a = [1, 2, 3, 4]
>>> id(a)
1075751756
>>> a[0] = 1
>>> id(a)
1075751756
```

- Oggetti immutabili il cui valore non può essere modificato senza creare un nuovo oggetto

```
>>> a = 5
>>> id(a)
135533752
>>> a = 3
>>> id(a)
135531768
```

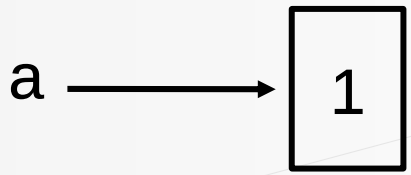
Viene creato un nuovo oggetto e ad a viene assegnata la reference del nuovo oggetto (nuovo binding)

Mutabili e Immutabili

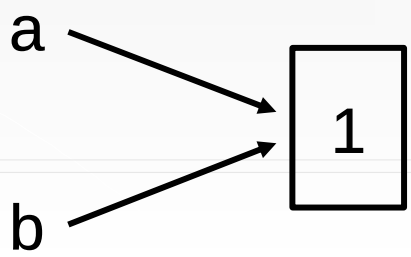
Tipo	Mutabile ?
Numeri	No
Stringhe	No
Liste	Si
Dizionari	Si

Assegnamento - 5

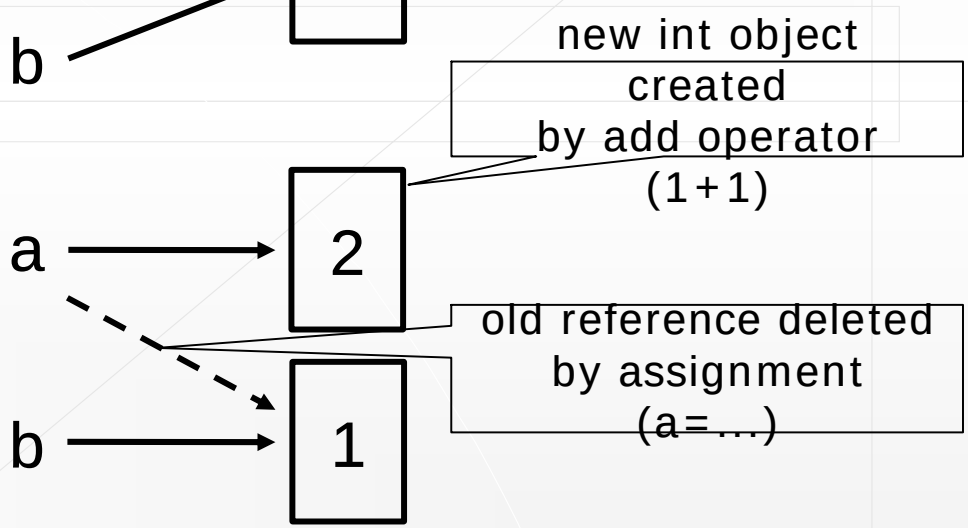
a = 1



b = a



a = a+1



Passaggio di parametri

- In Python il passaggio di parametri avviene per indirizzo: i parametri formali diventano degli alias dei parametri attuali

```
>>> a = [1,2]
>>> def swap(x):
...     temp = x[0]
...     x[0] = x[1]
...     x[1] = temp
...
>>> swap(a)
>>> print a
[2, 1]
```

La funzione ha modificato l'oggetto passato come parametro

Passaggio di parametri

- NB: se invece di modificare l'oggetto la funzione esegue un assegnamento (ovvero crea un nuovo binding), si interrompe il legame tra parametro formale e attuale che fanno ora riferimento a due celle distinte

```
>>> def f(x):
...     x = 1 # creo un nuovo binding. Perdo il
              collegamento con l'oggetto passato
              per parametro
...
>>> a = 1000
>>> f(a)
>>> print a # stampa 1000
```

Le modifiche non sono visibile al chiamante

Python e oltre

- Esistono numerose caratteristiche di Python che non sono state affrontate:
  - Supporto per il multi-thread (Java – style)
  - Servizi del sistema operativo
  - Protocolli di rete tcp, http, smtp, …
  - Reflection (simile a Java)
  - Parsing XML
  - Debugger
  - PyUnit (la versione Python di JUnit)
  - Librerie grafiche 2D (Tkinter, PyQt) e anche 3D (SDL)
  - …

Esempio: invio di una mail

```
import sys, smtplib
fromaddr = raw_input("From: ")
toaddrs = raw_input("To: ").split(',')
msg = ' '
while 1:
   line = sys.stdin.readline()
   if not line:
        break
   msg = msg + line

server = smtplib.SMTP('localhost')
server.sendmail(fromaddr, toaddrs, msg)
```