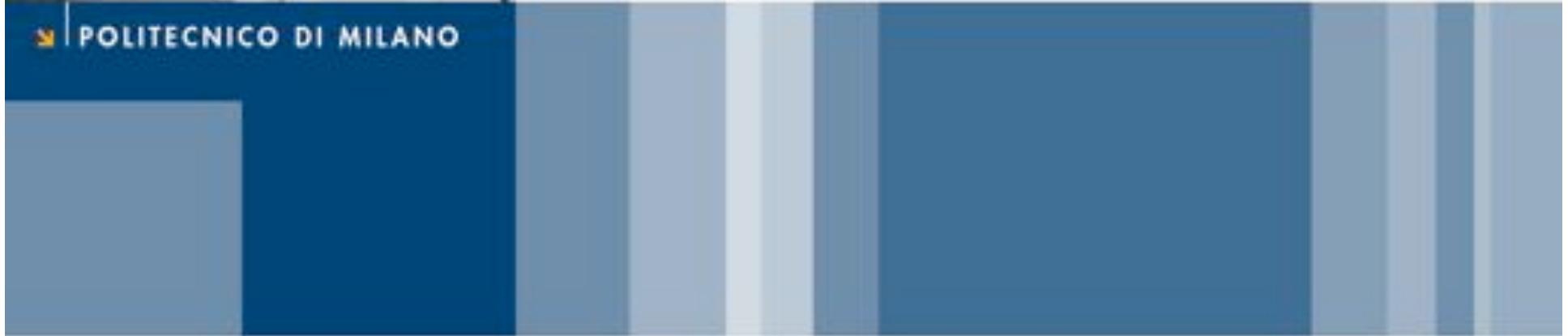




POLITECNICO DI MILANO



1. Introduzione a Python

Informatica 3

Andrea Mocci, mocci@elet.polimi.it

Slide gentilmente ereditate da Alessandro Campi (a sua volta ereditate da...)



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) è un fan...

Dove trovare informazioni...



- Sito ufficiale del linguaggio: interprete linux / win / MacOS, 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 introattivo)
<http://greenteapress.com/thinkpython>
- Google -> Python

Un linguaggio interpretato



- Python, a differenza di C/C++, è interpretato (anche se poi molte implementazioni lo compilano per motivi di efficienza):
- Si può 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 # concatenazione
Al fuoco! aiuto!
>>> ehm * 7 # ripetizione
'aiuto! aiuto! aiuto! aiuto! aiuto! aiuto! aiuto! '
>>> ehm * 0
'' # stringa vuota
```

Stringhe e indici



In Python le stringhe sono sequenze: si può 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
```

Però 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
-5	-4	-3	-2	-1



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

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

- Posso usare la ripetizione anche con le liste:
es.

```
>>> a = ["sono", "una", "lista"]  
>>> a * 0  
[]  
>>> a * 2  
['sono', 'una', 'lista', 'sono', 'una', 'lista']
```

append

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

```
>>> lista.append(3)  
>>> lista  
['una lista', 4, 6.2, 3]
```

- Posso concatenare con +

```
>>> a = ["sono", "una"]  
>>> b = ["lista"]  
>>> a+b  
['sono', 'una', 'lista']  
>>> #attenzione!!!  
>>> a.append(b)  
>>> a  
['sono', 'una', ['lista']]
```



Partiamo col ciclo while

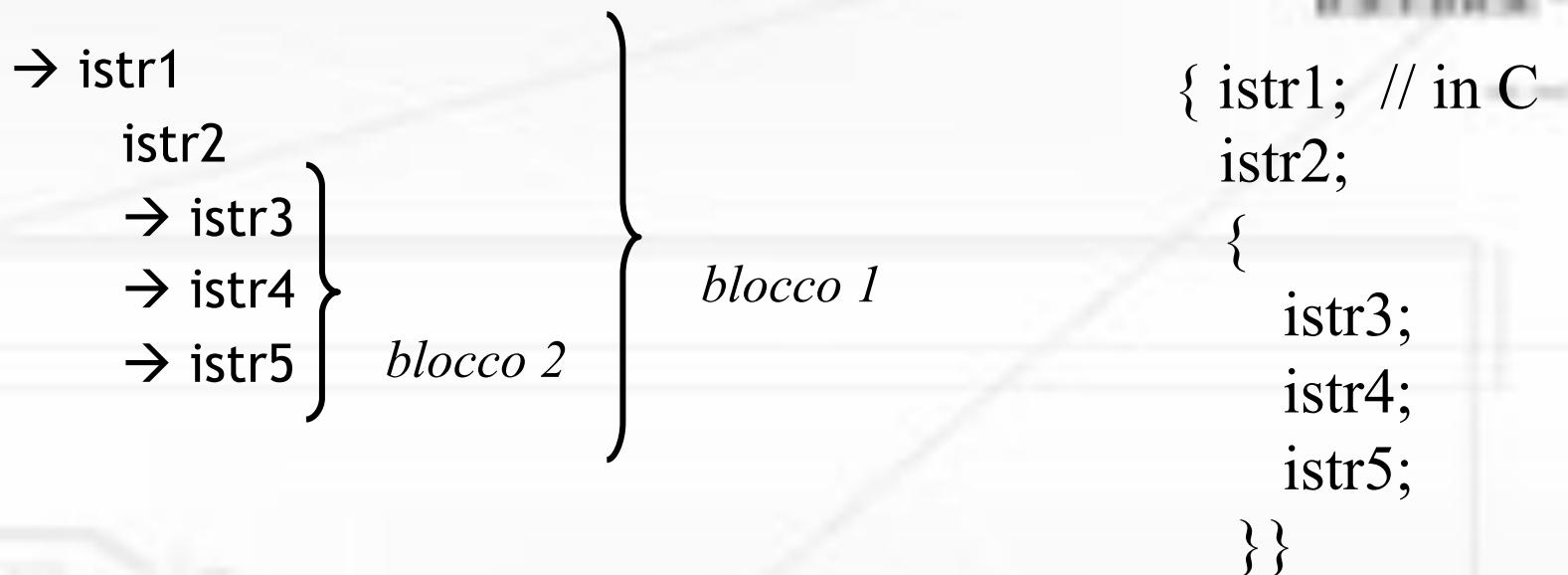
un semplice esempio: i numeri di Fibonacci

```
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) è delimitato per mezzo della *indentazione*!



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



- La forma più generale:

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

Ora il *for*

- Il for, a differenza che in 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 $\{(x,y) \mid x \in A, y \in B, x \neq y\}$ si puo` scrivere come

```
[ (x, y) for x in A for y in B if x != y]
```
- A proposito, (x,y) è un esempio di tupla (come in matematica) - in Python è una *sequenza immutabile*
 - si può usare come alternativa più efficiente alle liste, se non si devono modificare dati
- Attenzione! È comunque una *lista*, non un insieme (è un insieme totalmente ordinato che ammette più occorrenze dello stesso elemento)...

while



- Oltre al ciclo for, esiste 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)
```



- 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 risultato finale sia consistente)

I dizionari

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



```
>>> 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) è abbastanza classica in Python: sono metodi ed attributi con ruoli particolari - vedremo meglio nella parte sulla OO



- Python da 2.2 introduce le cosiddette nuove *classi* noi vedremo esclusivamente queste -- le altre rimangono per ragioni di compatibilità

Definizione di classe

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

costruttore

il progenitore di ogni classe è **object**

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

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

elementi "privati"

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

Ereditarietà



```
>>> 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 anche l'ereditarietà multipla

Overload

- Non posso fare overloading (se ho più 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



- È 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 è 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'è 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 cioè:
 - 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 è una reference a un istanza di P con in più 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 (ereditarietà)

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
```

Duck Typing

- “Se sembra un'anatra e fa il verso di un'anatra,
allora deve essere un'anatra”
 - “Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ()”.
 - Enfatizza le interfacce, non i tipi!
 - Il Duck-typing i test alla “instanceof”(che si possono fare usando type() o isinstance())
 - Di solito, si usano test tipo hasattr()

Assegnamento



- 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



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

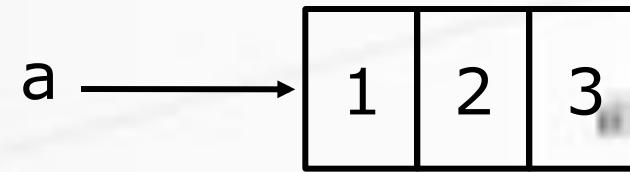
```
>>> a = [1,2,3]
>>> b = a
>>> id(a) # id(var) restituisce l'indirizzo (l-value) di var
135533752
>>> id(b)
135533752
```

- Si crea un alias: modificando a modifco anche b

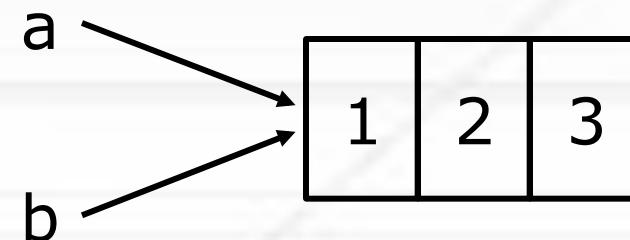
Assegnamento



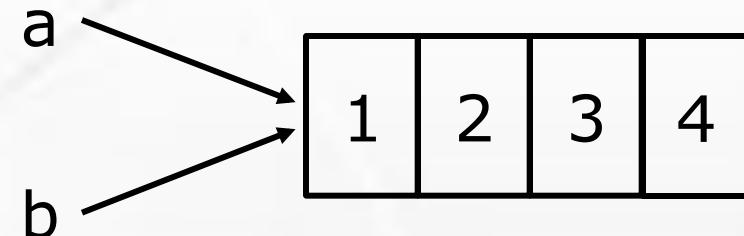
a = [1, 2, 3]



b = a



a.append(4)



Assegnamento



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

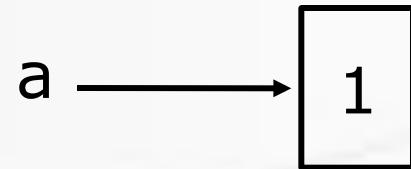


Tipo	Mutable ?
Numeri	No
Stringhe	No
Liste	Si
Dizionari	Si

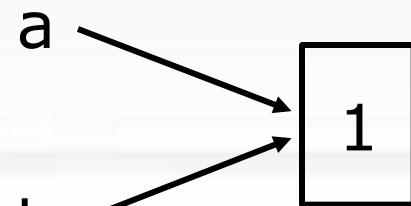
Assegnamento - 5



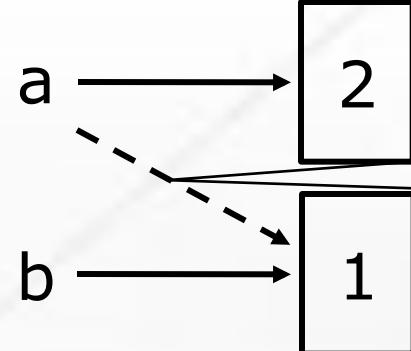
`a = 1`



`b = a`



`a = a+1`



new int object created
by add operator (1+1)

old reference deleted
by assignment (a=...)

Passaggio di parametri

- In Python il passaggio di parametri avviene per indirizzo: i parametri formali diventano 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 visibili 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)
```

Esercizio 16-02-05 es3



- Indicare l'output prodotto da questo programma
- python

```
def f(a, b, c, d):
    a.append(5)
    b = [3, 5]
    c[0] = 9
    d = 4

a = [0]
b = [1]
c = [2]
d = 3
f(a, b, c, d)

print a, b, c, d
```

Soluzione

- Il programma stampa [0, 5] [1] [9] 3
- Le Istruzioni:
 - 1) aggiunge un elemento alla lista del chiamante, modificandola
 - 2) L'assegnamento annulla il binding con la lista passata come parametro e ne effettua uno nuovo, lasciando inalterato l'oggetto originale.
 - 3) L'assegnamento modifica la lista del chiamante, cambiando il binding di un suo elemento.
 - 4) l'assegnamento annulla il binding con l'intero passato come parametro e ne effettua uno nuovo, lasciando inalterato l'oggetto originale.

Esercizio 29-04-04 es5



- Definire in Python la funzione map(f,lis) che, a partire da una funzione f() (che si suppone abbia un argomento) e da una lista seq = [e1 ... en] restituisca la lista [f(e1)...f(en)].

```
def map(f, lis):  
    s=[]  
    for i in lis  
        s.append(f(i))  
    return s
```

- Definire inoltre una funzione `accumulate(f, lis, zero)` che a partire da una funzione `f()` (che si suppone abbia due argomenti) e da una lista `seq = [e1 e2 ... en]` e dal valore zero restituisca il valore $f(\dots f(f(zero, e1), e2) \dots, en)$.

```
def accumulate(f, lis, zero):  
    res = zero  
    for i in lis :  
        res = f(res, i)  
    return res
```

- Facendo uso della funzione add (definita qui sotto), della funzione ithOdd(i) (pure definita qui sotto, restituisce l'i-simo numero dispari), delle funzioni map e accumulate, ed eventualmente di altri noti costrutti del linguaggio Python, definire la funzione squareBySum(n) che calcola il quadrato del numero n, assunto intero positivo, come somma dei primi n numeri dispari

(Es. 5 => [0 1 2 3 4] => [1 3 5 7 9]
=> 1+3+5+7+9 => 25).

```
def add(a, b):  
    return a+b  
  
def ithOdd(i):  
    return 2*i+1
```

```
def squareBySum(n):  
    return accumulate(add, map(ithOdd, range(n)), 0)
```