

*Alcuni lucidi sono tratti da una
ppresentazione di Marco Barisione*

Introduzione al linguaggio Python

Materiale su Python

- **Libri:**

- <http://www.python.it/doc/libri/>
(si consiglia in italiano
http://www.python.it/doc/Howtothink/HowToThink_ITA.pdf
La versione inglese, thinkpython, è più completa
- Dive Into Python (<http://it.diveintopython.org/>)
(più completo ma più complicato di thinkPython).

Esercizi Svolti e librerie:

<http://www.greenteapress.com/thinkpython/swampy/index.html>

(librerie per divertirsi e imparare con il python)

www.greenteapress.com/thinkpython/code

(esercizi da thinkPython, alcuni richiedono swampy)

<http://diveintopython.org/download/diveintopython-examples-5.4.zip>

(esempi tratti da diveintopython)

-
- È un linguaggio di programmazione:
 - Interpretato
 - Di alto livello
 - Semplice da imparare e usare
 - Potente e produttivo
 - Ottimo anche come primo linguaggio (molto simile allo pseudocodice)
 - Estensibile
 - Tipizzazione forte e dinamica (diversamente dalla shell di Linux)
 - Inoltre
 - È open source (www.python.org)
 - È multiplatforma
 - È facilmente integrabile con C/C++ e Java

È veramente usato da qualcuno?

- **RedHat**

anaconda (installazione), tool di configurazione grafici, log viewer

- **NASA**

- **www.google.com**

- **Industrial Light and Magic**

quelli che fanno gli effetti speciali per Star Wars...

- E molti altri ...

- E' incluso anche in Paint Shop Pro

- Ha centinaia di moduli (librerie) per ogni occorrenza, per esempio:

<http://www.catswhocode.com/blog/python-50-modules-for-all-needs>

Python: l'interprete interattivo

Python dispone di un interprete interattivo molto comodo e potente:

- Avvio: digitare `python` al prompt di una shell appare ora il prompt `>>>` pronto a ricevere comandi
- Possiamo ora inserire qualsiasi costrutto del linguaggio e vedere immediatamente l'output:

```
>>> 3+5
```

```
8
```

```
>>> print "Hello world!"
```

```
Hello world!
```

L'istruzione `print` stampa a video il risultato di un'espressione

L'interprete (1)

- L'interprete è un file denominato
 - “python” su Unix
 - “python.exe” su Windows, apre una console
 - “pythonw.exe” su Windows, non apre una console
- Se invocato senza argomenti presenta un'interfaccia interattiva
- I comandi si possono inserire direttamente dallo standard input
 - Il prompt è caratterizzato da “>>> ”
 - Se un comando si estende sulla riga successiva è preceduto da “...”
- I file sorgente Python sono file di testo, generalmente con estensione “.py”

L'interprete (2)

- Introducendo “#! /usr/bin/env python” come prima riga su Unix il sorgente viene eseguito senza dover manualmente invocare l'interprete
- Il simbolo “#” inizia un commento che si estende fino a fine riga
- Le istruzioni sono separate dal fine riga e non da “;”
 - Il “;” può comunque essere usato per separare istruzioni sulla stessa riga ma è sconsigliato
- Per far continuare un'istruzione anche sulla linea successiva è necessario inserire un “\” a fine riga
- Se le parentesi non sono state chiuse correttamente Python capisce che l'istruzione si estende anche sulla riga successiva

Alcuni concetti introduttivi

- Per capire il resto della presentazione serve sapere alcune cose
 - Le funzioni vengono chiamate come in C
 - `foo(5, 3, 2)`
 - “ogg.f()” è un metodo (spiegati nella parte sulle classi)
 - I metodi possono inizialmente essere considerati come delle funzioni applicate sull’oggetto prima del punto
 - Il seguente codice Python:
 - `"ciao".lower()`
 - può essere pensato equivalente al seguente codice C:
 - `stringa_lower("ciao");`

input e output

- L'istruzione "print" stampa il suo argomento trasformandolo in una stringa

```
>>> print 5
```

```
5
```

```
>>> print "Hello world"
```

```
Hello world
```

- A "print" possono essere passati più argomenti separati da un virgola. Questi sono stampati separati da uno spazio

```
>>> print 1, 2, "xxx"
```

```
1 2 xxx
```

- Se dopo tutti gli argomenti c'è una virgola non viene stampato il ritorno a capo

input e output

- Si può formattare l'output come il c:

```
>>> x=18; y=15
```

```
>>> print "x=%d y=%d\n" % (x,y)
```

```
x=18 y=15
```

- Per leggere un numero si usa input()

```
>>> x=input('Scrivi un numero:')
```

Da errore se non si inserisce un numero.

- Per leggere una stringa raw_input()

```
>>> x=raw_input('Scrivi il tuo nome:')
```

I numeri

- I numeri vengono definiti come in C:
 - “42” (intero, decimale)
 - “0x2A” (intero, esadecimale)
 - “052” (intero, ottale)
 - “0.15” (floating point, formato normale)
 - “1.7e2” (floating point, formato esponenziale)
 - Lo “0” iniziale o finale può essere omesso:
 - “.5” è “0.5”
 - “5.” è “5.0”
- Gli interi, se necessario, sono automaticamente convertiti in long (a precisione infinita)
 - **2 ** 64** → 18446744073709551616L

Gli operatori numerici

- Esistono gli stessi operatori del C, le parentesi possono essere usate per raggruppare:

```
>>> (5 + 3) * 2
```

```
16
```

```
>>> (6 & 3) / 2
```

```
1
```

- Esiste anche l'operatore elevamento “**”:

```
>>> 5 ** 2
```

```
25
```

- Non esistono “++” e “—”

Come in C, “3/2” è “1”

Altrimenti usare 3.0/2 (1.5)

3.0//2 è la divisione intera

Conversione fra numeri

- Esistono alcune utili funzioni di conversione fra i numeri:
 - “int(x[, base])” ritorna “x” convertito in intero. “base” è la base in cui “x” è rappresentato (il valore di default è “10”)
 - `int("13")` → 13
 - `int("13", 8)` → 11
 - `int("xxx")` → **Errore!**
 - “long(x[, base])” come “int” ma ritorna un “long”
 - “float(x)” ritorna “x” convertito in floating point
 - `float("13.2")` → 13.2
 - `float(42)` → 42.0

Variabili

- I nomi di variabili sono composti da lettere, numeri e underscore, il primo carattere non può essere un numero (come in C)
 - Sono validi:
 - “x”, “ciao”, “x13”, “x1_y”, “_”, “_ciao12”
 - Non sono validi:
 - “1x”, “x-y”, “\$a”, “àñÿô”
- Le variabili non devono essere dichiarate (Tipizzazione dinamica)
- Una variabile non può essere utilizzata prima che le venga assegnato un valore
- Ogni variabile può riferirsi ad un oggetto di qualsiasi tipo

Le variabili

Esempi:

```
x=5
```

```
nome="Marco"
```

- Sintetico

```
inizio, fine=2, 100
```

- type restituisce il tipo di una variabile

```
x=[ 5  3]
```

```
type (x)
```

```
>>> <type 'list'>
```

x è di tipo lista (verrà vista più avanti)

Ipython (1)

Scrivi ipython da una shell per farlo partire, exit () per uscire

Ipython può eseguire un programma python con il comando run (variabili rimangono in memoria)

In [1]: run prova.py

Il ? Alla fine da l'help

In [1]: print ?

Type restituisce il tipo di un qualsiasi oggetto

In [1]: x=15

In [2]: type(x)

Out[2]: int

Molti comandi Linux supportati: cd, cp, more, ls

Funzioni magiche (iniziano con %)

%magic mostra tutte le funzioni

%debug attiva il debug

%hist mostra la history

%pdoc int (mostra la documentazione per un comando o un tipo)

La % si può omettere

Ipython (2)

who # mostra le variabili

whos # Come who ma da anche i valori e altre info

In [14]: who

Interactive namespace is empty.

In [15]: shipname='Santa Maria'

In [16]: whos

Variable	Type	Data/Info
----------	------	-----------

shipname	str	Santa Maria
----------	-----	-------------

Ipython può creare un log della sessione Esempio:

logstart

a = 1+2

b = 3+4

who

logstop

more ipython_log.py

Scrivi log in ipython e premi il TAB (vale per tutti i comandi

In [1]: %log

%logoff %logon %logstart %logstate %logstop

In [2]: %logoff?

Assegnamento (1)

- L'assegnamento avviene attraverso l'operatore "="
- Non è creata una copia dell'oggetto:
 - `x = y` # si riferiscono allo stesso oggetto

- Esempio:

```
>>> x = [0, 1, 2]
```

```
>>> y = x
```

```
>>> x.append(3)
```

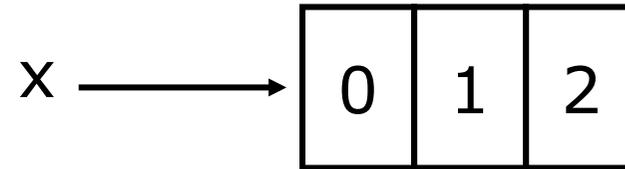
```
>>> print y
```

```
[0, 1, 2, 3]
```

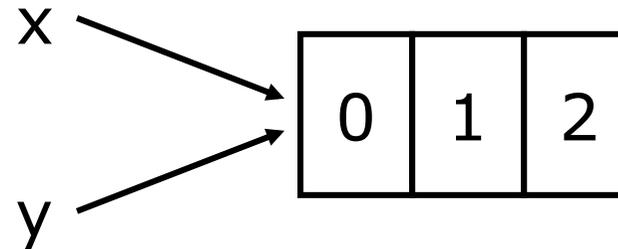
Assegnamento (2)

- Ecco quello che succede:

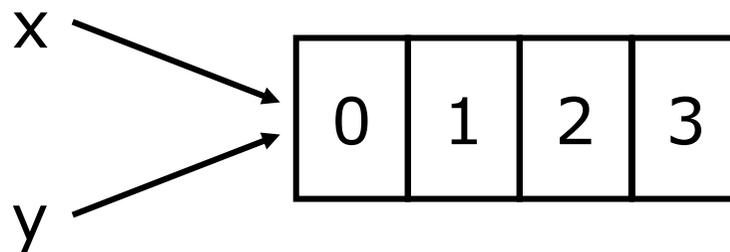
`x = [0, 1, 2]`



`y = x`

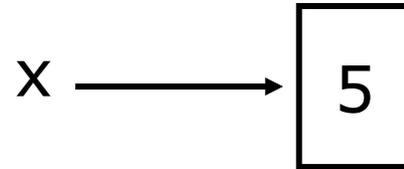


`x.append(3)`

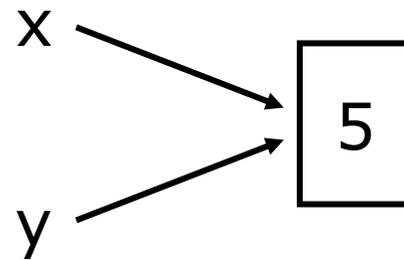


Assegnamento (3)

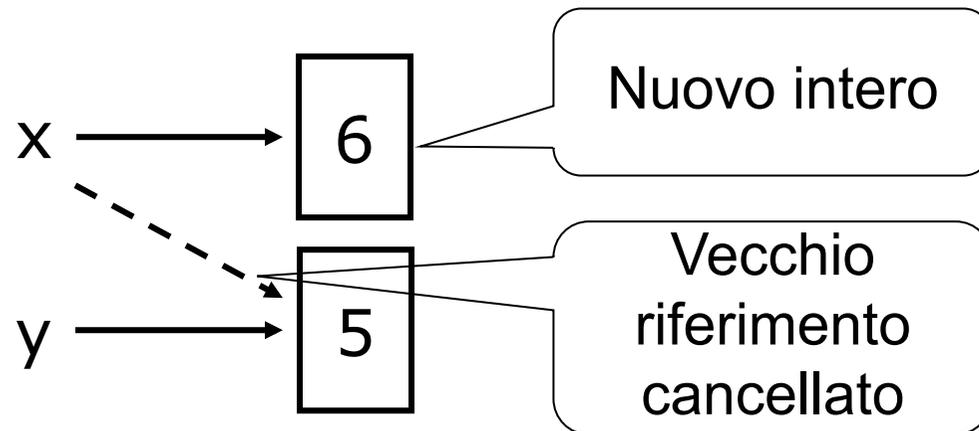
x = 5



y = x



x = x + 1



unpacking

- Un uso utile ed interessante dell'assegnamento è la possibilità di “scompattare” delle sequenze
 - $x, y = [1, 2]$ $\rightarrow x = 1, y = 2$
 - $x, y = \{ "a": 1, "b": 6 \}$ $\rightarrow x = "a", y = "b"$
 - $x, y, z = 1, 2, 3$ $\rightarrow x = 1, y = 2, z = 3$
- Può essere usato per scambiare i valori di due variabili
 - $x = 5$
 - $y = 3$
 - $x, y = y, x$ $\rightarrow x = 3, y = 5$
- Il numero di elementi a sinistra deve essere uguale al numero di elementi a destra

Augmented assignment statements

- Sono la combinazione di un assegnamento e di un operazione binaria
 - Ad esempio “x += 1” equivale a “x = x + 1”

- Gli operatori sono

+= -= *= /= //= %= **=

- Altri operatori:

and or not == !=

in is (li vediamo dopo)

a < b <= c < d # è valida in python

del

- L'istruzione "del" ha due usi differenti
 - "del x" cancella la variabile "x", cioè non si potrà più usare "x" senza avergli prima assegnato un nuovo valore
 - "del" non distrugge ciò a cui la variabile si riferisce come "free" in C!
 - "del seq[ind]" cancella l'elemento con indice/chiave "ind" da "seq"

```
li = [1, 2, 3]
```

```
del li[1]
```

→ li = [1, 3]

```
d = {"a": 1, "b": 6}
```

```
del d["a"]
```

→ d = {'b': 6}

```
l2 = [1, 2, 3, 4]
```

```
del l2[1:3]
```

→ l2 = [1, 4]

Vero e falso

- In Python esistono due variabili di tipo bool:
 - “True” uguale a 1 e “False” uguale a 0
- Ogni singolo tipo o classe può definire quando il suo valore è vero o falso
- Per i tipi predefiniti sono considerati falsi:
 - Il numero “0” o “0.0”
 - Una stringa vuota (“”)
 - “{}”, “[]”, “()”
- Gli altri valori sono considerati veri

Gli operatori di confronto

- Sono gli stessi del C
 - $1 == 3$ → Falso
 - $1 == 2 - 1$ → Vero
 - $1 != 2$ → Vero
 - $1 < 2$ → Vero
 - $1 > 3$ → Falso
 - $1 >= 1$ → Vero
- Se necessario vengono eseguite le necessarie conversioni intero → virgola mobile
 - $1 == 1.0$ → Vero
- Esiste anche l'operatore "<>" equivalente a "!=" ma obsoleto

Altri operatori di confronto

- “in”, vero se il primo operando è contenuto nel secondo
 - `5 in [1, 2, 3]` → Falso
 - `2 in [1, 2, 3]` → Vero
 - `"a" in {"x": 1, "a": 2}` → Vero
 - `"a" in "ciao"` → Vero
- “is”, vero se il primo operando è il secondo (non solo è uguale!)
 - Attualmente implementato come confronto fra le posizioni in memoria degli operandi (provate `id(x)`, da l’indirizzo di memoria)
 - Usato al posto di “==” per il confronto con “None” per motivi di prestazioni

Gli operatori booleani

- “not x” 0 se “x” è vero, “1” se è falso
- “x and y” vero se sia “x” sia “y” sono veri. Ritorna:
 - Se “x” è falso lo ritorna
 - Altrimenti ritorna “y”
 - `1 and 5` → 5 → Vero
 - `[] and 1` → [] → Falso
- “x or y” vero se almeno uno degli argomenti è vero
 - Se “x” è vero lo ritorna
 - Altrimenti ritorna “y”
 - `1 or 0` → 1 → Vero
 - `() or 0` → 0 → Falso
- Sia “and” sia “or” utilizzano la logica del corto circuito
 - Il secondo argomento non viene valutato se il risultato dell’operazione è già noto in base al solo primo argomento

if

- La sintassi di “if” è:

```
if condizione:
```

```
    ...
```

```
elif condizione_alternativa:
```

```
    ...
```

```
else:
```

```
    ...
```

- Sia la parte “elif” sia la parte “else” sono facoltative
- Può esserci un numero qualsiasi di “elif”
- Non sono necessarie le parentesi intorno all’espressione booleana
- Non sono possibili assegnamenti all’interno della condizione

L'indentazione

- Il raggruppamento è definito dall'indentazione
 - Non si usano parentesi graffe, coppie “begin”/“end” e simili
 - Obbliga a scrivere codice ordinato
 - Più naturale, evita i tipici problemi del C:

```
if (0)
    printf("Questo non viene eseguito");
    printf("Questo sì");
```

- Si possono usare spazi o tabulazioni
- È *fortemente* consigliato usare 4 spazi
 - Tutti gli editor decenti possono essere configurati per sostituire il TAB con 4 spazi

pass

- Come tradurre il seguente codice in Python?

```
if (a){}          /* Oppure "if (a);" */  
b = 12;
```

- In Python dopo un “if” deve esserci un blocco indentato

```
if a:  
    b = 12          # Dovrebbe essere indentato!
```

- Si usa quindi l’istruzione “pass” che non ha *nessun* effetto

```
if a:  
    pass          # Non fa nulla  
b = 12
```

while

- La sintassi è:

```
while condizione:  
    ...  
while i<10:  
    ...
```

- Si può uscire dal ciclo usando “break”
- Si può passare all’iterazione successiva usando “continue”
- Esempio:

```
while 1: # significa ciclo infinito  
    if condizione1:  
        continue  
    if condizione2:  
        break  
print 42
```

for

- La sintassi è:
`for variabile in iteratore:`
`...`
- “iteratore” può essere:
 - Una sequenza:
 - Liste
 - Tuple
 - Stringhe
 - Dizionari
 - Classi definite dall’utente
 - Un iteratore (nuovo concetto introdotto in Python 2.2)
- Si possono usare “continue” e “break” come per il “while”

Le stringhe

- Le stringhe sono racchiuse fra apici singoli o doppi e si utilizzano le stesse sequenza di escape del C

```
>>> 'Python'
```

```
'Python'
```

```
>>> print "Ciao\nmondo"
```

```
'Ciao'
```

```
'mondo'
```

- Si possono usare alcuni operatori visti per i numeri

```
>>> "ciao " + "mondo"      # concatenazione
```

```
'ciao mondo'
```

```
>>> "ABC" * 3              # ripetizione
```

```
'ABCABCABC'
```

Sottostringhe

c	i	a	o
0	1	2	3
-4	-3	-2	-1

- `"ciao"[1]` # carattere 1 → `"i"`
- `"ciao"[1:3]` # dall'1 al 3 escluso → `"ia"`
- `"ciao"[2:]` # dal 2 alla fine → `"ao"`
- `"ciao"[:3]` # fino al 3 escluso → `"cia"`
- `"ciao"[-1]` # l'ultimo carattere → `"o"`
- `"ciao"[:-2]` # fino al penultimo → `"ci"`
- `"ciao"[-1:1]` # non esiste → `""`
- Le stringhe sono immutabili (come i numeri):
 - `"ciao"[2] = "X"` → **Errore!**

Altri tipi di stringhe

- Le stringhe possono estendersi su più righe, in questo caso sono delimitate da tre apici singoli o doppi

```
"""Questa è una  
stringa su più righe"""
```

- For e stringhe

```
nome="Gianluigi"  
for c in nome:  
    print c
```

str e repr

- “str(x)” ritorna “x” convertito in stringa
- “repr(x)” ritorna una stringa che rappresenta “x”, normalmente più vicino a ciò che “x” è realmente ma meno “bello” di ciò che è ritornato da “str”

```
print "ciao\n\tmondo " → ciao
```

```
mondo
```

```
print repr("ciao\n\tmondo") → 'ciao\n\tmondo'
```

```
print 0.1, repr(0.1) → 0.1 0.10000000000000000001
```

```
print ogg → Questo è un oggetto
```

```
print repr(ogg) → <class C at 0x008A6570>
```

- “print” usa “str” automaticamente sul suo argomento

Stringhe, metodi

- “S.split([sep[, max]])” ritorna una lista contenente le parti di “S” divise usando i caratteri di “sep” come separatore. “max” è il numero massimo di divisioni eseguite. Il valore di default di “sep” è ogni spazio bianco
- “S.join(seq)” ritorna una stringa contenente gli elementi di “seq” uniti usando “S” come delimitatore
- “S.lower()” ritorna “S” convertito in minuscolo
- “S.upper()” ritorna “S” convertito in maiuscolo
- “S.find(what[, start[, end]])” ritorna il primo indice di “what” in “S[start, end]”. Se la sottostirnga non è trovata ritorna “-1”
- “S.rfind(what[, start[, end]])” come “find” ma a partire dal fondo
- “S.replace(old, new[, max])” ritorna una copia di “S” con “max” occorrenze di “old” sostituite con “new”. Il valore di default di “max” è tutte.
- “S.strip()” restituisce una copia di “S” senza gli spazi iniziali e finali
- “S.lstrip()”, “S.rstrip()” come “strip” ma eliminano rispettivamente solo gli spazi iniziali e finali

Formattazione di stringhe

- L'operatore “%” serve per formattare le stringhe in modo simile alla “printf” del C
 - `stringa % valore`
 - `stringa % (valore1, valore2, valore3)`
- Le stringhe di formato sono le stesse usate dal C
 - `"-%s-" % "x"` → `-x-`
 - `"%s%d" % ("x", 12)` → `x12`
- Per mantenere il simbolo “%” si inserisce “%%”
- Si può usare la forma “%(chiave)” per inserire le chiavi di un dizionario (struttura che verrà spiegata più avanti)
 - `"%(a)d,%(b)d" % {"a": 1, "b": 2}` → `1,2`

None

- “None” è una variabile molto importante con lo stesso ruolo di “NULL” in C
- In C “NULL” è uguale a “0”
 - Rischio di confusione
- In Python “None” è di un tipo non numerico
 - Non vi è rischio di confusione con altri tipi

importare moduli e usare funzioni

- Per importare il modulo delle funzioni matematiche:

```
import math
```

- Per usarlo:

```
math.sqrt(2) , math.log(5) , ecc.
```

Alcune funzioni built-in

- “range([start,] stop[, step])” ritorna una lista contenente gli interi in [“start”, “end”).
 - “step” è l’incremento, il valore di default è “+1”.
 - Il valore di default di “start” è “0”
 - Molto usato con i cicli “for”
 - `for i in range(5): print i`
- “len(seq)” ritorna il numero di elementi in “seq”
 - `len("ciao")` → 4
 - `len("x\0x")` → 3
 - `len([1, 2, 3])` → 3
 - `len({"a": 1, "b": 5})` → 2

Definire nuove funzioni (1)

- La sintassi è:

```
def funzione(arg1, arg2, opz1=val1, opz2=val2):  
    ...
```

- Non bisogna specificare il tipo ritornato
 - L'equivalente delle funzioni "void" del C sono funzioni che ritornano "None"
 - Una funzione può ritornare un oggetto di qualsiasi tipo
- Gli argomenti sono normali variabili e possono essere in qualsiasi numero
 - Se la funzione non accetta argomenti basta usare una lista di argomenti vuota, ad esempio:

```
def foo():  
    ...
```

Definire nuove funzioni (2)

- Gli argomenti opzionali possono non essere specificati dal chiamante, in questo caso assumono il valore di default
- Le variabili all'interno della funzione non sono visibili dall'esterno
- Esempio di utilizzo di una funzione:

```
>>> def foo(x, y, z=42, k=12):  
...     print x, y, z, k  
...  
>>> foo(5, 3, k=9)  
5 3 42 9
```

doc-string

- Le doc-string o stringhe di documentazione sono stringhe nella prima riga della funzione con lo scopo di documentarla

```
def foo():
```

```
    "Documentazione di foo"
```

- È possibile accedere alla doc-string con l'attributo `“__doc__”` della funzione
 - `print foo.__doc__` → Documentazione di foo
- Usata da tool per generare la documentazione
- Usata dalla funzione `“help”`
 - `“help(foo)”` stampa informazioni su `“foo”`

return

- La sintassi è:
`return [valore_di_ritorno]`
- Se il valore di ritorno viene omesso viene ritornato “None”
- Se il flusso del programma esce dalla funzione senza aver trovato un’istruzione “return” viene ritornato “None”
- Esempio:

```
def somma(a, b):  
    return a + b
```

Liste

- Contengono elementi anche eterogenei
- Sono implementate usando array e non liste
- Per creare una lista si usano le parentesi quadre, gli elementi sono delimitati da virgole

```
>>> [1, 2, "ciao"]
```

```
[1, 2, 'ciao']
```

- Stessi operatori delle stringhe ma sono mutabili

```
– [1] + [3, 6] → [1, 3, 6]
```

```
– [1, 0] * 3 → [1, 0, 1, 0, 1, 0]
```

```
– [2, 3, 7, 8][1:3] → [3, 7]
```

```
– [2, 3, 7, 8][:2] → [2, 3]
```

```
– [1, 2, 3][0] = 5 → [5, 2, 3]
```

Liste, metodi

- Ecco i metodi più usati:
 - “L.append(obj)”, aggiunge “obj” fondo
 - “L.extend(list)”, aggiunge in fondo gli elementi di “list”
 - “L.insert(index, obj)”, aggiunge “obj” prima di “index”
 - “L.pop([index]”, rimuove l’elemento in posizione “index” e lo ritorna, il valore di default di “index” è -1
 - “L.remove(value)”, cancella la prima occorrenza di “value”
 - “L.reverse()”, inverte la lista
 - “L.sort()”, ordina la lista
- Tutti “in place”, viene modificata la lista, non viene ritornata una nuova lista
- **ATTENZIONE** + rispetto ad append crea una nuova lista

Dizionari

- Associano ad una chiave un valore
- Creati nella forma “{chiave1: val1, chiave2: val2}”
 - {"nome": "Mario", "cognome": "Rossi"}
- L'accesso e l'inserimento di elementi avviene come per le liste
 - {"a": 1, "b": 2}["a"] → 1
 - {"a": 1, "b": 2}["X"] → **Errore!**
 - {}["X"] = 2 → {'X': 2}
- Le chiavi devono essere immutabili
- Le chiavi non vengono tenute ordinate!

Dizionari, metodi

- I metodi principali dei dizionari sono:
 - “D.clear()” elimina tutti gli elementi dal dizionario
 - “D.copy()” restituisce una copia di “D”
 - “D.has_key(k)” restituisce 1 se “k” è nel dizionario, 0 altrimenti. Si può usare anche l’operatore “in”
 - “D.items()”, “D.keys()”, “D.values()” restituiscono rispettivamente:
 - Una lista con le tuple “(chiave, valore)”
 - La lista delle chiavi
 - La lista dei valori
 - “D.update(D2)” aggiunge le chiavi e valori di “D2” in “D”
 - “D.get(k, d)” restituisce “D[k]” se la chiave è presente nel dizionario, “d” altrimenti. Il valore di default di “d” è “None”

Introspection

type restituisce il tipo di una variabile

```
x=[ 5  3]
type (x)
>>> <type 'list'>
```

dir ritorna tutti i metodi di un modulo oppure del tipo associato ad una variabile:

```
x=[ 5  3]
dir(x)
#ritorna tutti i metodi di una lista
dir(math)
#ritorna tutti i metodi di math
```

Accesso ai file (1)

- I file vengono gestiti in modo molto semplice e simile al C
- “open(nomefile[, modo])” apre “nomefile” in modalità “modo” (“r” è il valore di default) e ritorna un oggetto di tipo “file”
- I modi sono gli stessi del C
- I metodi principali degli oggetti file sono:
 - “read([n])” ritorna “n” byte dal file. Se “n” è omesso legge tutto il file
 - “readline()” ritorna una riga
 - “readlines()” ritorna una lista con le righe rimanenti nel file
 - “write(str)” scrive “data” sul file

Accesso ai file (2)

- “writelines(list)” scrive tutti gli elementi di “list” su file
- “close()” chiude il file (richiamato automaticamente dall’interprete)
- “flush()” scrive su disco i dati presenti in eventuali buffer
- “seek(offset[, posiz])” muove di “offset” byte da “posiz”. I valori di posiz sono:
 - 0: dall’inizio del file (valore di default)
 - 1: dalla posizione corrente
 - 2: dalla fine del file (“offset” è normalmente negativo)
- “tell()” ritorna la posizione corrente
- “truncate([n])” tronca il file a non più di “n” byte. Il valore di default è la posizione corrente

global

- L'assegnamento all'interno della funzione assegna il valore ad una variabile locale

```
x = 5
```

```
def f():
```

```
    x = 42 # x è nuova variabile locale!
```

- Con “global nome_var” si indica all'interprete che “nome_var” è globale e non locale

```
x = 5
```

```
def f():
```

```
    global x
```

```
    x = 42 # x è la variabile globale
```

lambda (1)

- “lambda” serve a creare funzioni anonime di *una sola* istruzione, viene ritornato il valore dell’espressione
- La sintassi è

```
lambda arg1, arg2, arg3: istruzione
```
- Usata spesso per implementare callback

```
def f(callback):  
    for i in range(10): print callback(i)  
f(lambda n: n ** 2)
```
- Sconsigliata, si può sempre sostituire con funzioni
 - Nell’esempio precedente:

```
def cb(n): return n ** 2  
f(cb)
```

lambda (2) e sorted

- Esempi di uso
- Reduce applica la funzione a una lista

```
L=[1,2,3,4,5,6]
```

```
z=reduce(lambda x,y:x*y,L)
```

```
z=120 (prodotto della lista)
```

Ordinare liste multiple (esempio per il secondo elemento):

```
lis=[["marco",18],["John",15],["Antony",24]]
```

```
lista=sorted(lis,key=lambda x:x[1])
```

oppure

```
lista=sorted(lis,key=itemgetter(1)) oppure
```

```
lista.sort(key=itemgetter(1))
```