

# Informatica

*Corso di Laurea in Scienze Geologiche*

## Seconda Parte

Ugo Dal Lago



ALMA MATER STUDIORUM  
UNIVERSITÀ DI BOLOGNA



Anno Accademico 2015-2016

## Sezione 1

Prepariamoci...

# Alcune Caratteristiche di Python

- ▶ Python è un linguaggio **di alto livello**.
  - ▶ In altre parole, nella gerarchia delle macchine astratte, si trova in posizione alta.
  - ▶ Da un punto di vista concettuale, è più vicino ai problemi e alle applicazioni che alla macchina fisica.
- ▶ Python è un linguaggio **general-purpose**.
  - ▶ Non ci sono troppi limiti teorici agli algoritmi che si possono scrivere e ai problemi che si possono risolvere.
- ▶ Python è un linguaggio **interpretato**, e non compilato.
  - ▶ Ciò impatta leggermente le prestazioni dei programmi, ma in maniera certo non determinante nella maggior parte dei settori.

# Perché Python?

- ▶ Perché è relativamente **semplice** da imparare.
  - ▶ I programmi risultano molto compatti.
  - ▶ Il fatto di essere interpretato ci permette di sperimentare in modo semplice e di ottenere semplicemente *feedback*.
- ▶ Perché esistono una moltitudine di **librerie**, ossia di programmi già scritti da altri, che spesso permettono di semplificare molto la risoluzione dei problemi.
- ▶ Perché è uno dei linguaggi più **utilizzati** dalla comunità delle scienze applicate (e quindi da *biologi*, *geologi*, etc.).
  - ▶ Questa non è certo la motivazione principale, però!

# Storia di Python

- ▶ Nasce nel 1990, ad opera dell'informatico olandese *Guido van Rossum*
- ▶ Ha “*cambiato pelle*” più volte:
  - ▶ Fino al 2000 era un linguaggio di nicchia.
  - ▶ Nel 2000, con l'avvento di **Python 2.0**, cominciò una fase in cui moltissime comunità cominciarono a scrivere librerie per i domini più disparati.
  - ▶ Nel 2008, venne introdotto **Python 3.0**, che risolse molti piccoli problemi della versione 2.
    - ▶ Noi, però, seguendo il libro di testo, useremo **Python 2.7**.
- ▶ Oggi **Python** è un linguaggio di estremo successo, utilizzato da milioni di sviluppatori.

# Sperimentare con Python

- ▶ Per cominciare a sperimentare con il linguaggio Python, basta utilizzare un portale web dedicato, ossia

`http://www.pythontutor.com/`

- ▶ È possibile visualizzare in modo preciso cosa succede durante l'esecuzione di semplici programmini Python.
- ▶ L'uso di `pythontutor` è fortemente consigliato durante lo studio domestico.
- ▶ Se `pythontutor` è assolutamente sufficiente per passare l'esame scritto, non lo è per il progetto.

# Installare Python

- ▶ Quando si cominciano a scrivere programmi di una certa complessità, `pythontutor.com` non basta.
- ▶ L'interprete `Python` è disponibile in tutti i principali sistemi operativi, ivi inclusi LINUX, MACOSX, e WINDOWS.
- ▶ Interagiremo con l'interprete `Python` attraverso un'applicazione chiamata IDLE.
- ▶ Se utilizzare LINUX e MACOSX, l'interprete `Python`(e IDLE) sono già preinstallati.
- ▶ Se invece utilizzare WINDOWS, potete installare il programma recuperandolo, ad esempio, da qui:

`http://www.python.it/download/`

- ▶ Scaricate e installate la versione 2.7.10.

## Sezione 2

Pronti, Via!



# Struttura dei Programmi Python

- ▶ Un programma **Python** non è altro che una sequenza di **comandi** e **definizioni**.
- ▶ Un comando (altrimenti detto **statement**) è una istruzione che impartisco all'interprete.
  - ▶ Posso per *esempio* chiedergli di stampare a video qualcosa, con il comando **print**
- ▶ Un esempio di programma potrebbe quindi essere

```
print 'Hello, World!'
print 'Good morning!'
print 'Have a nice day!'
```

# Oggetti Scalari

- ▶ Gli **oggetti** sono le entità fondamentali che i programmi manipolano.
- ▶ Ogni oggetto ha un **tipo**, che identifica il tipo di operazioni che i programmi possono fare con oggetti di quel tipo.
- ▶ Gli **oggetti scalari** sono quegli oggetti che sono indivisibili, e che quindi non hanno struttura interna.
- ▶ Python dispone di quattro tipi di oggetti scalari:
  1. Il tipo `int`, ovvero il tipo dei numeri interi. Ad esempio, i letterali `3`, `-215`, o `321000`.
  2. Il tipo `float`, ovvero il tipo dei numeri reali. Ad esempio, `3.0`, `pyt3.17`, o `1.6E7`.
    - ▶ Queste sono in realtà *approssimazioni finite* di numeri reali.
  3. Il tipo `bool`, ovvero il tipo degli oggetti booleani, ovvero `True` e `False`
  4. il tipo `None`, ovvero un tipo con un solo oggetto.
- ▶ Quelli che abbiamo dato, sono (da un punto di vista sintattico) letterali. Come aggregarli?

# Espressioni

- ▶ Possiamo formare **espressioni** aggregando tra di loro letterali, **operatori** (e altre categorie sintattiche) nel modo naturale.
- ▶ Occorre, ovviamente, che gli operatori vengano applicati a letterali ed espressioni rispettando il relativo *tipo*.
- ▶ Ogni espressione potrà essere *valutata* ad un oggetto, che chiameremo il suo **valore**.

- ▶ **Esempi:**

```
>>> 3 + 2
```

```
5
```

```
>>> 3.0 + 2.0
```

```
5.0
```

```
>>> 3 != 2
```

```
True
```

- ▶ Il tipo del valore cui valuta un'espressione lo possiamo calcolare con l'operatore **type**

# Operatori Aritmetici

Significato	
+	Addizione
-	Sottrazione
*	Moltiplicazione
//	Divisione Intera
/	Divisione
%	Resto della Divisione Intera
**	Elevamento a Potenza

# Operatori di Confronto

Significato	
==	Uguaglianza
!=	Disuguaglianza
<	Minore Stretto
<=	Minore o Uguale
>	Maggiore Stretto
>=	Maggiore o Uguale

# Operatori Logici

Significato	
<b>and</b>	Congiunzione: il risultato è <b>true</b> se e solo se <i>entrambi</i> gli operandi sono <b>true</b>
<b>or</b>	Disgiunzione: il risultato è <b>true</b> se e solo se <i>almeno uno</i> degli operandi è <b>true</b>
<b>not</b>	Negazione: il risultato è <b>true</b> se e solo se l'operando (che è unico!) è <b>false</b>

# Operatori e Tipi

- ▶ Che tipo devono avere gli **operandi**?
- ▶ E che tipo avrà il **risultato**?
- ▶ **Python** ha delle regole di tipo abbastanza permissive, proprio per la sua natura di linguaggio senza tipi statici.
- ▶ Ad esempio, negli operatori aritmetici:
  - ▶ Se entrambi gli operandi hanno tipo `int`, il risultato è `int`
  - ▶ Se almeno uno degli operandi ha tipo `float`, allora il risultato ha tipo `float`.
- ▶ Quando si applica, ad esempio, un operatore logico ad un letterale di tipo numerico, **Python** cerca in tutti i modi di risolvere la questione **senza** sollevare un errore di semantica statica.
  - ▶ In questo modo però la semantica dinamica diventa più *complessa*, spesso imprevedibile.
  - ▶ Cercate di evitare situazioni come queste.

## Esempi di Espressioni — I

```
>>> (3.2)+4
```

```
7.2
```

```
>>> (7//2)==(7/2)
```

```
True
```

```
>>> (7//2)==(7.0/2.0)
```

```
False
```

```
>>> (3==4) or (3!=(5+3))
```

```
True
```

```
>>> ((7%2)+2**3)==9.0
```

```
True
```

```
>>> 3==3.0
```

```
True
```

```
>>> 3<3.0
```

```
False
```



## Esempi di Espressioni — II

```
>>> type(3*5.0)
<type 'float'>
>>> type(3*5)
<type 'int'>
>>> 7.3%2
1.2999999999999998
>>> 7%2
1
>>> 3.0 and 2.4
2.4
>>> not 3.9
False
```

# Tre Modi per Interagire con l'Interprete Python

## 1. Shell

- ▶ La shell di **Python** è un semplice *valutatore* di comandi ed espressioni.
- ▶ Li esegue uno a uno, *mostrandone* il risultato (se si tratta di un'espressione).
- ▶ La shell è contraddistinta dalla sequenza `>>>`. La sua presenza ci indica che la shell è pronta a valutare un comando o un'espressione.
- ▶ Abbiamo già usato la shell negli esempi precedenti!
- ▶ Una volta valutata, un'espressione è in linea di principio *persa*.

## 2. File

- ▶ Appena cominciamo ad avere a che fare con comandi complessi, o addirittura con programmi, la shell diventa inadeguata.
- ▶ Risulta quindi molto utile scrivere i nostri programmi in un normale *file di testo*.
- ▶ IDLE mette a disposizione un editor per file di questo tipo.
- ▶ Comandi da eseguire in sequenza andranno a finire in *righe successive* del file.

# Tre Modi per Interagire con l'Interprete Python

## 3 PythonTutor

- ▶ È equivalente all'utilizzo dei file.
- ▶ Non occorre però aver installato l'interprete Python, né un editor di testo.
- ▶ Basta utilizzare un qualunque browser, andare all'URL <http://www.pythontutor.com>, e scegliere “Visualize Code”.

# Sommario

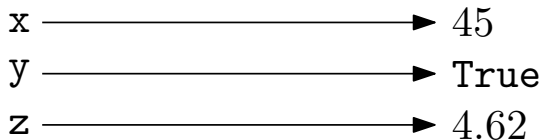
- ▶ Finora, gli *unici comandi* a nostra disposizione sono:
  - ▶ Le **espressioni**, che denotano un valore, ma che se usate come comandi non producono nessun effetto.
  - ▶ L'istruzione **print**, che stampa a video il valore denotato da un'espressione.

# Sommario

- ▶ Finora, gli *unici comandi* a nostra disposizione sono:
  - ▶ Le **espressioni**, che denotano un valore, ma che se usate come comandi non producono nessun effetto.
  - ▶ L'istruzione **print**, che stampa a video il valore denotato da un'espressione.
- ▶ L'esecuzione di ciascuno comando in un programma è quindi completamente indipendente dagli altri.
- ▶ Non c'è modo di far comunicare parti distinte di un programma.
- ▶ In realtà, **Python** (come tutti i linguaggi di programmazione imperativi), mette a disposizione il concetto di *variabile* e di *stato interno*.

# Variabili e Stato Interno — I

- ▶ Le **variabili Python** non sono nient'altro che sequenze di caratteri, ossia *nomi*.
  - ▶ Alcune di tali sequenze, dette **parole chiave**, non possono essere usate come nomi.
  - ▶ L'unico esempio di parola chiave visto finora è **print**.
- ▶ In ogni momento, lo **stato interno** di un programma Python in esecuzione, è dato da un'associazione tra *alcune* variabili e degli *oggetti*.
- ▶ Ad esempio:



## Variabili e Stato Interno — II

- Consideriamo il seguente codice:

```
pi = 3
raggio = 13
area = pi * (raggio**2)
raggio = 11
print pi
print area
print raggio
```

- Che valori stamperà?

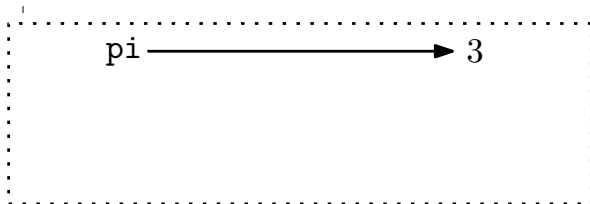
```
➔ pi = 3  
  raggio = 13  
  area = pi * (raggio**2)  
  raggio = 11
```





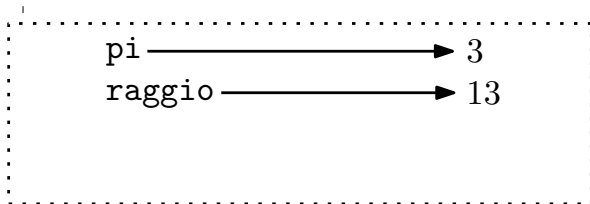
## Variabili e Stato Interno — III

```
    pi = 3  
    ➔ raggio = 13  
    area = pi * (raggio**2)  
    raggio = 11
```



## Variabili e Stato Interno — III

```
pi = 3  
raggio = 13  
➔ area = pi * (raggio**2)  
raggio = 11
```



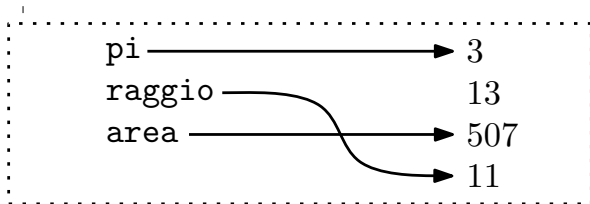
## Variabili e Stato Interno — III

```
pi = 3  
raggio = 13  
area = pi * (raggio**2)  
➔ raggio = 11
```

```
pi —————> 3  
raggio —————> 13  
area —————> 507
```

## Variabili e Stato Interno — III

```
pi = 3  
raggio = 13  
area = pi * (raggio**2)  
raggio = 11
```



# Assegnamenti

- ▶ Nel programma di esempio abbiamo usato l'**assegnamento**, uno dei costrutti fondamentali di Python.
- ▶ L'assegnamento è l'unico vero modo che abbiamo per modificare lo stato interno e in particolare per associare un nuovo valore ad una variabile.
- ▶ Le variabili possono essere sovrascritte, e il vecchio valore è *perso*.
- ▶ La **sintassi** è:

$$\langle \text{assegnamento} \rangle \rightarrow \langle \text{variabile} \rangle = \langle \text{espressione} \rangle$$

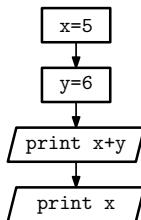
- ▶ La semantica dinamica dell'assegnamento  $\mathbf{x}=\mathbf{e}$  è, come ci si può immaginare, la seguente:
  1. Valuto l'espressione  $\mathbf{e}$ , ottenendo un valore  $V$ .
  2. Creo un'oggetto nello stato interno che contiene  $V$  (anche se ce ne sono altri).
  3. Creo un'associazione tra  $\mathbf{x}$  e  $V$  (eliminando eventuali frecce originanti da  $\mathbf{x}$ ).

## Istruzioni Condizionali — I

- ▶ I programmi che abbiamo visto finora sono *puramente sequenziali*.
  - ▶ Ogni istruzione determina in modo univoco quale sia la successiva.

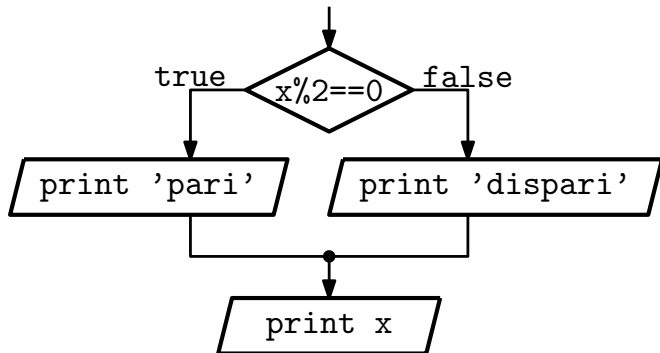
# Istruzioni Condizionali — I

- ▶ I programmi che abbiamo visto finora sono *puramente sequenziali*.
  - ▶ Ogni istruzione determina in modo univoco quale sia la successiva.
- ▶ Possiamo rappresentare il programma attraverso un **diagramma di flusso**, in questo modo evidenziando la sua struttura interna.
- ▶ Ad esempio, il diagramma di flusso di un programma sequenziale potrebbe avere la struttura seguente:



## Istruzioni Condizionali — II

- Potremmo avere bisogno, invece, di un diagramma di flusso diverso:



- Come procediamo?
- C'è bisogno di un comando in grado di gestire l'esecuzione condizionale di comandi.

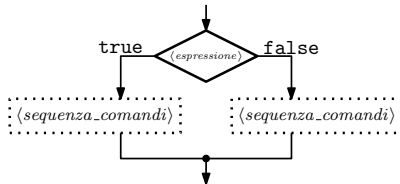


# Comando if-else

- La sintassi è la seguente:

```
if <espressione>:  
    <sequenza_comandi>  
else :  
    <sequenza_comandi>
```

- L'**indentazione** è assolutamente rilevante, e va rispettata.
- Il corrispondente diagramma di flusso sarebbe il seguente:



# Pari e Dispari

```
if (x%2 == 0):  
    print 'Pari'  
else:  
    print 'Dispari'  
print x
```

## Un Esempio Più Complesso

```
if (x%2 == 0):  
    if (x%3 == 0):  
        print 'Divisibile per 2 e 3'  
    else:  
        print 'Divisibile per 2 ma non per 3'  
else:  
    if (x%3 == 0):  
        print 'Divisibile per 3 ma non per 2'  
    else:  
        print 'Non divisibile per 3 e per 2'
```

# Gestire Molti Casi

- ▶ `if` e `else` sono quindi **parole chiave**, e non possono essere usate come nomi di variabili.
- ▶ Esiste anche un'altra parola chiave, ossia `elif` che serve per rendere il codice più compatto.
- ▶ **Esempio:**

```
if x < y and x < z:  
    print 'x è il minore'  
elif y < z:  
    print 'y è il minore'  
else:  
    print 'z è il minore'
```

## Il Tipo `str`

- ▶ Il tipo `str` è il tipo delle **stringhe**, ossia delle **sequenze di caratteri**.
- ▶ Gli oggetti di questo tipo, dunque, sono oggetti *non* scalari.
- ▶ I letterali di tipo stringa sono sequenze di caratteri delimitati:
  - ▶ Dall'apostrofo, ossia `'`;
  - ▶ Dalle virgolette, ossia `"`.
- ▶ **Esempio:**

```
x = 'abc'
y = "def"
print x
print y
if x != y:
    print 'Sono diverse'
else:
    print 'Sono uguali'
```

# Stringhe e Operatori

- ▶ Alcuni operatori aritmetici possono essere utilizzati sulle stringhe, e diventano in questo modo molto utili:
  - ▶ L'operatore + diventa un modo per **concatenare** stringhe.
  - ▶ L'operatore \* diventa un modo per **ripetere** una stringa un certo numero di volte.
- ▶ Gli operatori di confronto <, >, <= e >= diventano un modo per confrontare le stringhe secondo l'ordine alfabetico.
- ▶ **Esempio:**

```
>>> 'abc'+'def'
'abcdef'
>>> 'abc'*3
'abccabccab'
>>> 3*'abc'
'abccabccab'
>>> 'abc'<'def'
True
>>> 'aaaac'<='aaaa'
False
>>> 'a'<'A'
False
```

# Indicizzazione

- ▶ L'**indicizzazione** permette di estrarre singoli caratteri da una stringa.
- ▶ L'indice del primo carattere della stringa è 0, e va crescendo spostandosi verso la fine della stringa.
- ▶ I numeri negativi sono utilizzati per accedere alla stringa a partire dagli ultimi caratteri.
- ▶ La sintassi è  $\langle espressione \rangle [\langle indice \rangle]$ .
- ▶ **Esempio:**

```
>>> x='abcdef'
>>> x[0]
'a'
>>> x[4]
'e'
>>> x[6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
>>> x[-2]
'e'
```

# Slicing

- ▶ La lunghezza di una stringa può essere calcolata tramite la funzione `len`.
- ▶ L'indicizzazione permette di accedere a *singoli* caratteri della stringa.
- ▶ Talvolta, invece, occorre estrarre *porzioni* di una certa stringa, e questo è ciò che permette di fare lo **slicing**.
- ▶ La sintassi dello slicing è

$$\langle \text{espressione} \rangle [\langle \text{indice\_sinistro} \rangle : \langle \text{indice\_destro} \rangle]$$

dove la porzione di stringa selezionata è quella:

- ▶ Il *primo* carattere è quello di indice  $\langle \text{indice\_sinistro} \rangle$ ;
  - ▶ L'*ultimo* carattere è quello di indice  $\langle \text{indice\_destro} \rangle$ .
- ▶ Uno degli indici può essere *omesso*. In tal caso:
    - ▶ Se ometto  $\langle \text{indice\_sinistro} \rangle$ , il frammento selezionato inizia con il carattere di indice 0;
    - ▶ Se ometto  $\langle \text{indice\_destro} \rangle$ , il frammento selezionato termina con il carattere di indice `len(a)-1`, dove **a** è la stringa.



## Slicing — Esempio

```
>>> x = 'abcdef'
>>> x[0:1]
'a'
>>> x[0:3]
'abc'
>>> x[:4]
'abcd'
>>> x[3:]
'def'
>>> len(x)
6
>>> x[1:-1]
'bcde'
```

# Input

- ▶ I programmi **Python** che abbiamo scritto finora permettono di interagire con le periferiche di *output*, ma non con le periferiche di *input* (ad esempio la **tastiera**)
- ▶ **Python** mette a disposizione una varietà di funzioni che permettono di chiedere, a chi sta usando il programma, di inserire una stringa.
- ▶ Noi utilizzeremo solo **raw\_input**.
  - ▶ La stringa che passiamo a **raw\_input** è il messaggio che l'utente vedrà prima di inserire la stringa.
  - ▶ Il valore inserito dall'utente verrà poi restituito dalla funzione.

## Input — Esempio

```
>>> name = raw_input('Inserisci il tuo nome: ')
Inserisci il tuo nome: Pinco Pallino
>>> risposta = raw_input('Ma il tuo nome è '+name+' ? ')
Ma il tuo nome è Pinco Pallino ? Certo
>>> print risposta
Certo
>>> numero = raw_input('Inserisci un numero: ')
Inserisci un numero: 345
>>> print numero
345
>>> print type(numero)
<type 'str'>
```

# Conversioni di Tipo Esplicite

- ▶ E' possibile convertire, ad esempio, una stringa *che contenga un numero intero* nel numero stesso?
- ▶ Python permette di eseguire queste operazioni tramite appositi **operatori di conversione esplicite**.
- ▶ La sintassi:

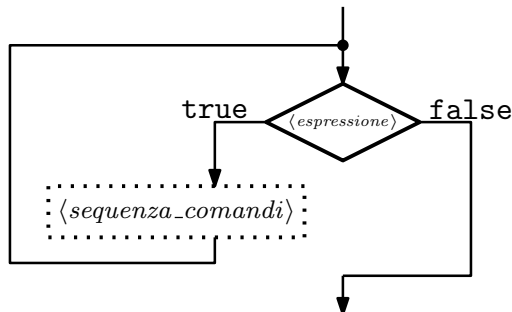
$$\langle \text{espressione} \rangle \rightarrow \langle \text{tipo} \rangle (\langle \text{espressione} \rangle)$$

- ▶ **Esempio:**

```
>>> a='123'  
>>> print int(a)  
123  
>>> b=int(a)  
>>> print b*3.2  
393.6  
>>> c=str(b)  
>>> print c  
123
```

# Iterazione

- ▶ Nel linguaggio **Python** è possibile scrivere programmi che si comportino come il seguente diagramma di flusso:



- ▶ Tali programmi si chiamano **programmi iterativi**.
- ▶ A differenza di tutti i programmi visti finora, nei programmi iterativi *lo stesso comando* può essere eseguito più volte.
  - ▶ In tal caso, ovviamente, lo stato interno sarà diverso.

# Il Comando `while`

- ▶ Il comando `while` è uno dei modi in cui Python permette di scrivere programmi iterativi.
- ▶ La sua sintassi è la seguente:

```
while <espressione>:  
    <sequenza_comandi>
```

- ▶ L'indentazione è, come nel caso del comando `if`, rilevante.
- ▶ La semantica dinamica è la seguente:
  - ▶ Valuta l'espressione.
  - ▶ Se tale espressione valuta a `true`, *esegui* la sequenza di comandi, e al termine *ricomincia* il procedimento daccapo.
  - ▶ Se tale espressione valuta a `false`, passa all'istruzione successiva.

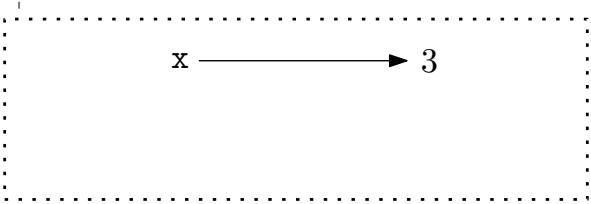
## Un Esempio

```
→ x = 3
   ris = 0
   itman = x
   while (itman != 0):
       ris = ris + x
       itman = itman - 1
   print ris
```



## Un Esempio

```
    x = 3  
    ➔ ris = 0  
    itman = x  
    while (itman != 0):  
        ris = ris + x  
        itman = itman - 1  
    print ris
```

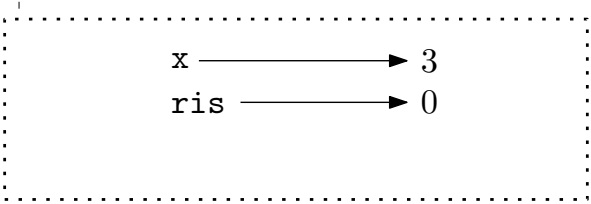


A diagram enclosed in a dashed rectangular box. It shows a variable 'x' on the left, followed by a horizontal arrow pointing to the right, which terminates at the number '3'.



## Un Esempio

```
x = 3
ris = 0
➔ itman = x
while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```

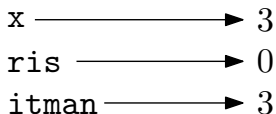


A dashed rectangular box containing two lines of text. The first line shows the variable 'x' followed by a horizontal arrow pointing to the number '3'. The second line shows the variable 'ris' followed by a horizontal arrow pointing to the number '0'.

x	→	3
ris	→	0

## Un Esempio

```
x = 3
ris = 0
itman = x
➔ while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```

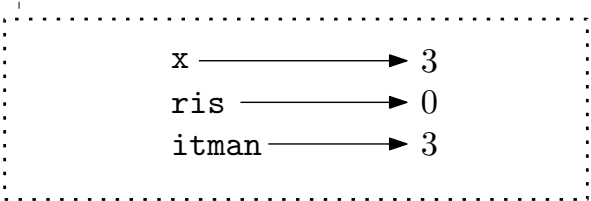


A diagram enclosed in a dashed rectangular box. It shows three variables on the left, each with a horizontal arrow pointing to its current value on the right. The variable 'x' has an arrow pointing to '3'. The variable 'ris' has an arrow pointing to '0'. The variable 'itman' has an arrow pointing to '3'.

x	→	3
ris	→	0
itman	→	3

## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ➔    ris = ris + x
        itman = itman - 1
print ris
```



A dashed rectangular box containing a diagram of variable values. On the left, the variables 'x', 'ris', and 'itman' are listed vertically. To the right of each variable is a horizontal arrow pointing to its current value: 'x' points to '3', 'ris' points to '0', and 'itman' points to '3'.

x	→	3
ris	→	0
itman	→	3

## Un Esempio

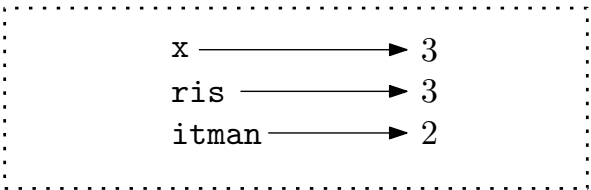
```
x = 3
ris = 0
itman = x
while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```



x	→	3
ris	→	3
itman	→	3

## Un Esempio

```
x = 3
ris = 0
itman = x
➡ while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```



A dashed rectangular box containing three lines of text. Each line consists of a variable name, a horizontal arrow pointing to the right, and a numerical value. The variables are x, ris, and itman. The values are 3, 3, and 2 respectively.

x	→	3
ris	→	3
itman	→	2

## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ➔    ris = ris + x
        itman = itman - 1
print ris
```

x	—————→	3
ris	—————→	3
itman	—————→	2

## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```



x	→	3
ris	→	6
itman	→	2

## Un Esempio

```
x = 3
ris = 0
itman = x
➔ while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```

A diagram enclosed in a dashed rectangular box. It shows three variables: 'x', 'ris', and 'itman'. Each variable is followed by a horizontal arrow pointing to its current value. 'x' points to 3, 'ris' points to 6, and 'itman' points to 1.

x	→	3
ris	→	6
itman	→	1



## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ➔    ris = ris + x
        itman = itman - 1
print ris
```

x	—————→	3
ris	—————→	6
itman	—————→	1

## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ris = ris + x
    ➔ itman = itman - 1
print ris
```

1

x	—————→	3
ris	—————→	9
itman	—————→	1

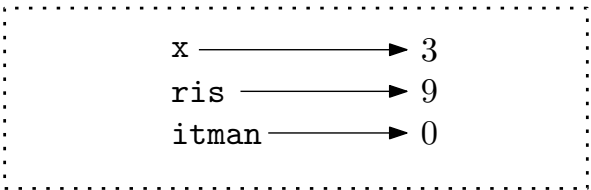
## Un Esempio

```
x = 3
ris = 0
itman = x
➔ while (itman != 0):
    ris = ris + x
    itman = itman - 1
print ris
```

```
x → 3
ris → 9
itman → 0
```

## Un Esempio

```
x = 3
ris = 0
itman = x
while (itman != 0):
    ris = ris + x
    itman = itman - 1
➔ print ris
```



x	→	3
ris	→	9
itman	→	0

## Sezione 3

### Rudimenti di Algoritmica

# Il Problema

- ▶ All'inizio del corso, avevamo parlato del problema di calcolare la radice quadrata di un numero.
- ▶ In questa parte, diamo alcune soluzioni al problema, e a variazioni dello stesso.
  - ▶ Il problema di per sè non è cruciale, ma è un ottimo pretesto per parlare di alcune argomenti particolarmente interessanti.

## Prima Soluzione

```
x=int(raw_input('Inserisci un intero: '))
ris = 0
while ris**3 < abs(x):
    ris = ris + 1
if ris**3 != abs(x):
    print x, 'non e un cubo'
else:
    if x < 0:
        ris = -ris
    print 'Il cubo di ', x, ' e ', ris
```

# Il Ciclo for

- ▶ Finora abbiamo visto solo un modo per costruire programmi iterativi, ossia il ciclo **while**.
- ▶ Ne esiste anche un altro, spesso più semplice da usare.
- ▶ La sua sintassi è la seguente:

**for**  $\langle \text{variabile} \rangle$  **in**  $\langle \text{sequenza\_valori} \rangle$ :  
     $\langle \text{sequenza\_comandi} \rangle$

- ▶ La semantica dinamica è la seguente:
  1. Calcola la  $\langle \text{sequenza\_valori} \rangle$ .
  2. Esegui  $\langle \text{sequenza\_comandi} \rangle$  un numero di volte pari alla **lunghezza della sequenza** costruita al punto precedente.
    - ▶ In ciascuna di queste iterate, la variabile  $\langle \text{variabile} \rangle$  prende uno dei valori in  $\langle \text{sequenza\_valori} \rangle$
    - ▶ Se in  $\langle \text{sequenza\_comandi} \rangle$  viene eseguita l'istruzione **break**, l'esecuzione del ciclo si interrompe.



# Come Costruire Sequenze?

- ▶ Il modo più semplice per costruire sequenze è la funzione predefinita **range**, che prende in input tre valori interi:
  - ▶ Il primo, **start** è il valore **di partenza**;
  - ▶ Il secondo, **stop** è il valore **finale**;
  - ▶ Il terzo, **step** indica **di quanto** aumentano i valori lungo la sequenza (e può essere negativo).
- ▶ Il valore **step** può essere omissso, e in tal caso vale 1.
- ▶ **Esempi:**

```
>>> range(0,4,2)
```

```
[0, 2]
```

```
>>> range(1,6,1)
```

```
[1, 2, 3, 4, 5]
```

```
>>> range(1,6)
```

```
[1, 2, 3, 4, 5]
```

```
>>> range(3,8,-1)
```

```
[]
```

```
>>> range(8,3,-1)
```

```
[8, 7, 6, 5, 4]
```

## Ancora sul Ciclo for

- ▶ I cicli `for` possono essere annidati arbitrariamente, e a differenza del ciclo `while`, tutto questo risulta in codice elegante e ben strutturato.

- ▶ **Esempio:**

```
for i in range(1,4):  
    for j in range(0,i):  
        print j
```

- ▶ Il ciclo `for` può essere anche utilizzato per scorrere gli elementi di una stringa.
  - ▶ In tal caso, la sequenza di valori non è nient'altro che la stringa stessa.

- ▶ **Esempio:**

```
totale = 0  
for c in '123456789':  
    totale = totale + int(c)  
print totale
```

## Seconda Soluzione

```
x=int(raw_input('Inserisci un intero: '))
for ris in range(0, abs(x)+1):
    if ris**3 >= abs(x):
        break
if ris**3 != abs(x):
    print x, 'non e un cubo'
else:
    if x < 0:
        ris = -ris
    print 'Il cubo di ', x, ' e ', ris
```

# Enumerazione Esaustiva

- ▶ I programmi che abbiamo appena scritto usano una tecnica molto semplice per esplorare lo spazio di ricerca, ossia l'**enumerazione esaustiva**.
- ▶ Man mano che il valore di  $x$  cresce, la quantità di tempo necessario a trovare la soluzione cresce molto rapidamente.
  - ▶ Se  $x$  è un intero di una decina di cifre, il tempo necessario ad eseguire il programma è enorme.
  - ▶ **Morale:** l'enumerazione esaustiva spesso è la tecnica più naturale, ma occorre comunque essere al corrente dei rischi che si corrono utilizzandola.

# Un Problema Diverso

- ▶ Supponiamo ora di voler calcolare la radice quadrata di un *qualunque* numero intero, senza limitarci ai quadrati perfetti.
- ▶ La strada che percorreremo sarà quella di trovare delle **approssimazioni** del numero cercato.
- ▶ Useremo, come nei casi precedenti, il metodo dell'enumerazione esaustiva.

# Prima Soluzione

```
x=int(raw_input('Inserisci un intero: '))
epsilon = 0.01
step=epsilon**2
numtent = 0
ris = 0.0
while (abs(ris**2 - x) >= epsilon) and (ris*ris <= x):
    ris += step
    numtent += 1
print 'numtent =', numtent
if abs(ris**2 - x) >= epsilon:
    print 'La ricerca ha avuto esito negativo'
else:
    print ris, ' e una buona approssimazione di ', x
```

## Alcune Osservazioni

- ▶ L'algoritmo che abbiamo appena descritto è corretto sia quando  $x$  è superiore a 1, sia quando è inferiore a 1.
- ▶ Il numero di volte in cui il ciclo **while** è eseguito è nell'ordine di  $x/\text{step}$ .
- ▶ Se  $x$  è molto grande, è possibile che la ricerca abbia esito negativo.
  - ▶ Si può diminuire **epsilon**, ma in tal caso il numero di iterazioni aumenta considerevolmente
- ▶ Un modo per diminuire considerevolmente il numero di iterazioni consiste nel rimpiazzare l'enumerazione esaustiva con un'algoritmo più "furbo", ovvero la cosiddetta **ricerca dicotomica**.
  - ▶ In questo modo il numero di iterazioni diventa nell'ordine del *logaritmo* di  $x/\text{step}$ , funzione che cresce molto lentamente con  $x$ .

## Seconda Soluzione

```
x=int(raw_input('Inserisci un intero: '))
epsilon = 0.01
numGuesses = 0
low = 0.0
high = max(1.0, x)
ris = (high + low)/2.0
while abs(ris**2 - x) >= epsilon:
    print 'low =', low, 'high =', high, 'ris =', ris
    numtent += 1
    if ris**2 < x:
        low=ris
    else:
        high = ris
    ris = (high + low)/2.0
print 'numtent =', numtent
print ris, 'is close to square root of', x
```



## Sezione 4

### Astrazione e Funzioni

# Astrazione e Riutilizzo

- ▶ I costrutti di **Python** che abbiamo studiato finora sono solo una **minima** parte di quelli disponibili nel linguaggio.
- ▶ Ciò non vuol dire che non si possa scrivere nulla con essi, anzi.
  - ▶ Siamo già in presenza di un linguaggio cosiddetto *Turing-completo*.
- ▶ Ciò che non abbiamo ancora visto, però, è un meccanismo che permetta l'**astrazione** e il **riutilizzo del codice**.
  - ▶ Potremmo trovarci a dover risolvere problemi *molto simili* ma diversi, le cui soluzioni algoritmiche sono esse stesse simili.
  - ▶ Potremmo aver bisogno di scrivere un programma complesso in cui *in due o più situazioni distinte* sia necessario risolvere lo stesso sotto-problema
- ▶ In entrambi i casi ci troveremmo a dover **duplicare** codice.

# Funzioni

- ▶ Abbiamo già visto svariate funzioni predefinite del linguaggio Python.
  - ▶ **Esempi:** `max`, `len`, `type`.
- ▶ È possibile, nel linguaggio Python, *definire nuove funzioni*.
- ▶ La sintassi per la **definizione di funzioni** è la seguente:

```
def <nome_funzione>(<lista_parametri>):  
    <sequenza_comandi>
```

- ▶ `<lista_parametri>` non è nient'altro che una lista di nomi di variabili.
- ▶ **Esempio:**

```
def max(x,y):  
    if x<y:  
        return y  
    else  
        return x
```

- ▶ Il comportamento della funzione `max` quello che ci aspettiamo.

# Parametri Formali e Attuali

- ▶ Supponiamo di avere una definizione di funzione come la seguente:

```
def abc(x,y,z):  
    return z
```

- ▶ Dopo aver dichiarato **abc**, è possibile **chiamarla** come se fosse una funzione predefinita:

```
>>> abc(3,4,5)  
5
```

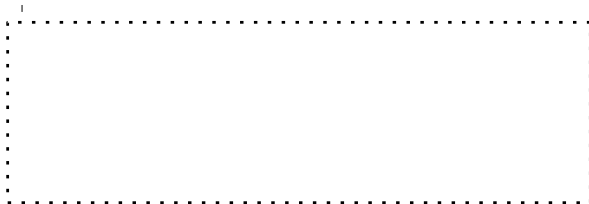
- ▶ Le variabili **x**, **y**, **z** sono dette **parametri formali** di **abc**.
- ▶ I valori **3**, **4**, **5** sono invece detti **parametri attuali** della *chiamata* ad **abc**.
- ▶ La sequenza di comandi di cui si compone la definizione di funzione è un programma **Python** (indentato).
  - ▶ L'unica **eccezione** è la possibilità dell'istruzione **return**, attraverso la quale è possibile restituire un valore al chiamante.

# Semantica Dinamica delle Funzioni

1. I *parametri attuali*, ossia le espressioni che troviamo contestualmente alla chiamata, vengono **valutati**, ottenendo altrettanti valori, ciascuno relativo ad un *parametro formale*
2. La prossima istruzione da eseguire diventa la *prima* della funzione *chiamata*.
3. La sequenza di istruzioni della funzione chiamata **viene eseguita**, finché non si incontra un'istruzione **return** oppure finché non si esegue l'ultima istruzione della funzione chiamata.
  - Nel primo caso il **valore di ritorno** è quello passato all'istruzione **return**, mentre nel secondo è **None**.
4. Il controllo **ritorna** all'istruzione che aveva effettuato la chiamata, e il valore con cui sostituire la funzione è il valore di ritorno.

# Diagrammi di Stato in Presenza di Funzioni

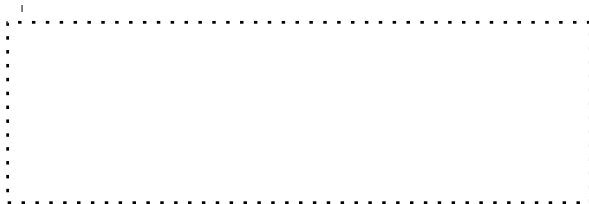
```
➔ def add(x,y):  
    z = x+y  
    return z  
  
a=3  
print add(a,8)
```



# Diagrammi di Stato in Presenza di Funzioni

```
def add(x,y):  
    z = x+y  
    return z
```

```
→ a=3  
print add(a,8)
```

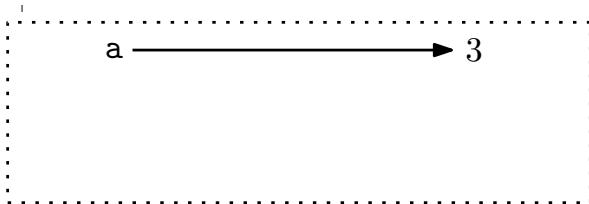


# Diagrammi di Stato in Presenza di Funzioni

```
def add(x,y):  
    z = x+y  
    return z
```

```
a=3
```

```
→ print add(a,8)
```





# Diagrammi di Stato in Presenza di Funzioni

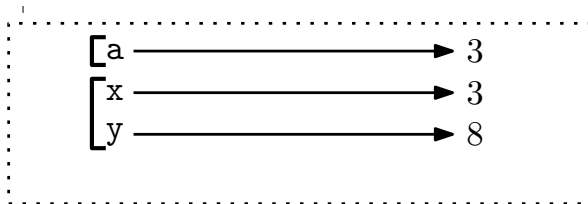
➔ `def add(x,y):`

`z = x+y`

`return z`

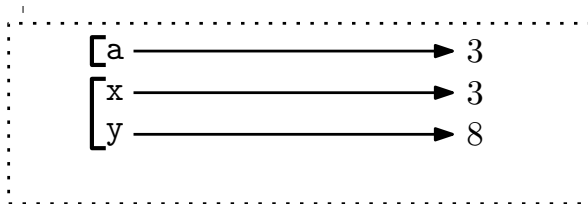
`a=3`

`print add(a,8)`



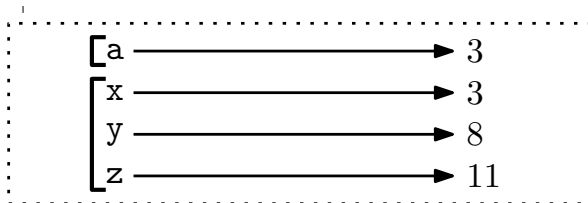
# Diagrammi di Stato in Presenza di Funzioni

```
def add(x,y):  
    → z = x+y  
    return z  
  
a=3  
print add(a,8)
```



# Diagrammi di Stato in Presenza di Funzioni

```
def add(x,y):  
    z = x+y  
    →    return z  
a=3  
print add(a,8)
```

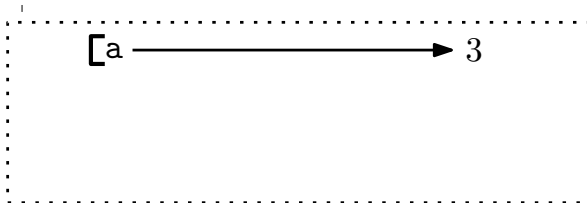


# Diagrammi di Stato in Presenza di Funzioni

```
def add(x,y):  
    z = x+y  
    return z
```

```
a=3
```

```
→ print add(a,8)
```



## Parametri Nominali — I

- ▶ Partiamo da un **esempio**:

```
def stNome(nome, cognome, inv):  
    if inv:  
        print cognome + ', ' + nome  
    else:  
        print nome, cognome
```

- ▶ Le seguenti chiamate alla funzione `stNome` sono tutte equivalenti

```
>>> stNome('Pinco', 'Pallino', False)  
Pinco Pallino  
>>> stNome('Pinco', 'Pallino', inv=False)  
Pinco Pallino  
>>> stNome('Pinco', cognome='Pallino', inv=False)  
Pinco Pallino  
>>> stNome(nome='Pinco', cognome='Pallino', inv=False)  
Pinco Pallino
```

- ▶ Parliamo in questo caso di **parametri nominali**.

## Parametri Nominali — II

- ▶ Non è possibile far seguire un *parametro nominale* da un *parametro non nominale*.
  - ▶ Altrimenti Python non saprebbe proprio a che parametro formale far corrispondere il parametro attuale!
- ▶ I parametri nominali vengono spesso usati in combinazione con i cosiddetti **parametri di default**.

- ▶ **Esempio:**

```
def stNome(nome, cognome, inv = False):  
    if inv:  
        print cognome + ', ' + nome  
    else:  
        print nome, cognome
```

- ▶ In questo modo anche la chiamata  
stNome('Pinco', 'Pallino') ha un effetto identico alle precedenti.

# Regole di Scoping — I

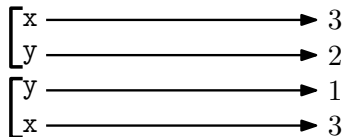
- Consideriamo il seguente **esempio**:

```
def f(x):  
    y = 1  
    x = x + y  
    print 'x =', x  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print 'z =', z  
print 'x =', x  
print 'y =', y
```

- Come **si comporta** il programma quando eseguito?
  - In particolare, che succede quando una variabile *locale* (ossia definita all'interno di una funzione) è identica ad una variabile *globale*?

## Regole di Scoping — II

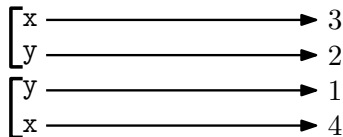
```
def f(x):  
    y = 1  
    → x = x + y  
    print 'x =', x  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print 'z =', z  
print 'x =', x  
print 'y =', y
```





## Regole di Scoping — II

```
def f(x):  
    y = 1  
    x = x + y  
→   print 'x =', x  
    return x  
  
x = 3  
y = 2  
z = f(x)  
print 'z =', z  
print 'x =', x  
print 'y =', y
```



## Regole di Scoping — III

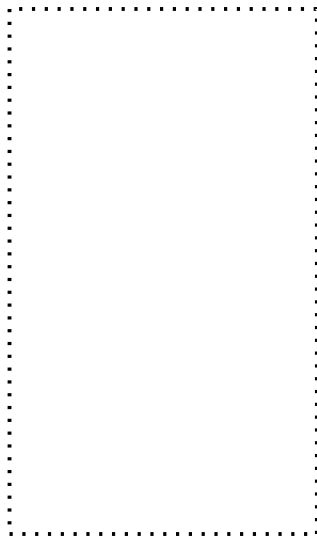
- ▶ In presenza di chiamate a funzioni, le variabili nel diagramma di stato sono organizzate in una pila di **frame**.
  - ▶ Ciascun frame corrisponde ad una chiamata di funzione.
  - ▶ La stessa variabile può comparire in più di un frame.
  - ▶ Ogniqualvolta una funzione è chiamata, viene creato un nuovo frame “in cima” alla pila.
  - ▶ Al termine dell'esecuzione di una funzione, il relativo frame è eliminato dalla pila.

## Regole di Scoping — III

- ▶ In presenza di chiamate a funzioni, le variabili nel diagramma di stato sono organizzate in una pila di **frame**.
  - ▶ Ciascun frame corrisponde ad una chiamata di funzione.
  - ▶ La stessa variabile può comparire in più di un frame.
  - ▶ Ogniqualvolta una funzione è chiamata, viene creato un nuovo frame “in cima” alla pila.
  - ▶ Al termine dell’esecuzione di una funzione, il relativo frame è eliminato dalla pila.
- ▶ Nel caso in cui la stessa variabile compaia in più di un frame, come si determina quello effettivamente in gioco?
  - ▶ Esistono delle regole precise, dette **regole di scoping statico**, che permettono di risolvere quest’ambiguità.
  - ▶ In presenza di più di un’occorrenza di una stessa variabile  $x$ , quella utilizzata è quella sintatticamente più “vicina” alla funzione correntemente in esecuzione.

## Regole di Scoping — IV

```
➔ def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

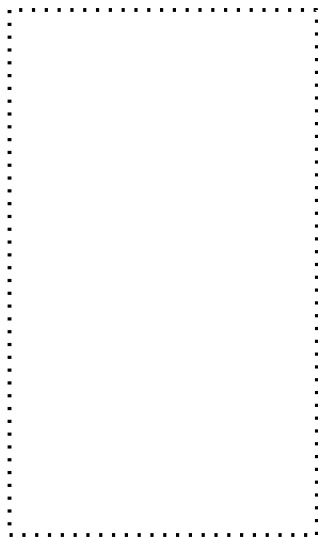


## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x
```

➔ x = 3

f(x)



## Regole di Scoping — IV

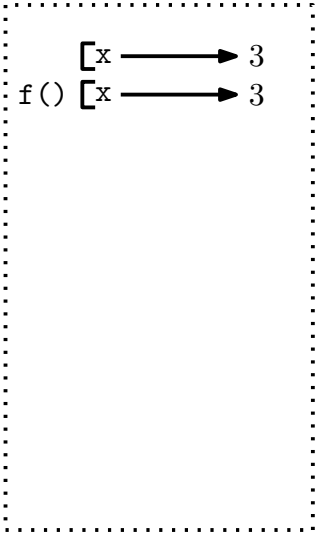
```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
  
x = 3
```

→ f(x)

[x → 3

## Regole di Scoping — IV

```
→ def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```



f() [x → 3  
[x → 3

## Regole di Scoping — IV

→

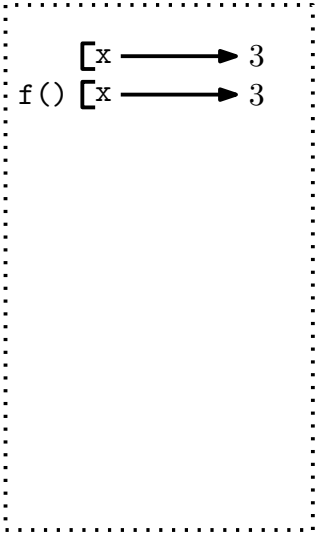
```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

[x → 3  
f() [x → 3



## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
→ x = x+1  
  print 'x =', x  
  h()  
  g()  
  print 'x =', x  
x = 3  
f(x)
```



f() [x → 3  
[x → 3

## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

f() [x → 3]  
[x → 4]

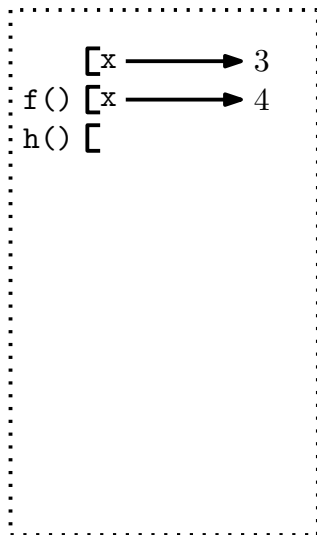
## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

$[x \longrightarrow 3$   
f()  $[x \longrightarrow 4$

## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```



## Regole di Scoping — IV

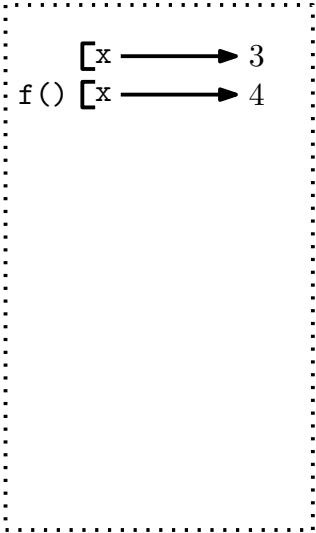
```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```



[x → 3  
f() [x → 4  
h() [z → 4

## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```



f() [x → 3]  
[x → 4]

## Regole di Scoping — IV

→

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

[x → 3  
f() [x → 4  
g() [

## Regole di Scoping — IV

→

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
x = 3  
f(x)
```

[x → 3  
f() [x → 4  
g() [x → 'abc'



## Regole di Scoping — IV

```
def f(x):  
    def g():  
        x = 'abc'  
        print 'x =', x  
    def h():  
        z = x  
        print 'z =', z  
    x = x+1  
    print 'x =', x  
    h()  
    g()  
    print 'x =', x  
→ x = 3  
f(x)
```

[x] → 3  
f() [x] → 4

# La Ricorsione — I

- ▶ Le **definizioni ricorsive** sono utilizzate nei domini più disparati, e in particolare in *matematica* e *informatica*.
- ▶ **Esempi:**
  - ▶ Una *matrioska* è:
    - ▶ Una bambola molto piccola;
    - ▶ oppure una bambola che si può aprire e che al suo interno contiene un'altra matrioska.
  - ▶ Il *fattoriale*  $n!$  di un numero naturale  $n$  può essere definito come segue:

$$n! = \begin{cases} 1 & \text{se } n = 0; \\ n \cdot (n-1)! & \text{se } n \geq 1. \end{cases}$$

- ▶ L' $n$ -esimo *numero di Fibonacci*  $\text{fib}(n)$  è definito come segue:

$$\text{fib}(n) = \begin{cases} 1 & \text{se } n \in \{0, 1\}; \\ \text{fib}(n-1) + \text{fib}(n-2) & \text{se } n \geq 2. \end{cases}$$

## La Ricorsione — II

- ▶ Come molti altri linguaggi di programmazione, Python permette di definire **funzioni ricorsive**, rendendo in questo modo più semplice e leggibile il codice.
- ▶ Una funzione **a** si dice **ricorsiva** se tra le sue istruzioni vi è una chiamata ad **a** stessa.
- ▶ **Esempio:**

```
def factI(n):  
    res=1  
    while n > 1:  
        res=res*n  
        n-=1  
    return res
```

```
def factR(n):  
    if n==1:  
        return n  
    else:  
        return n*factR(n-1)
```

- ▶ La semantica dinamica delle funzioni ricorsive è identica a quella di qualunque altra funzione.
  - ▶ **Nota bene:** potremo avere più di un frame relativo alla stessa funzione!

## La Ricorsione — III

- ▶ Un altro problema in cui la ricorsione può diventare molto utile è il **controllo di palindromia**.
- ▶ Osserviamo che una stringa **a** è palindroma *se e solo se*:
  - ▶ **a** è la stringa vuota;
  - ▶ oppure se il primo carattere di **a** è identico all'ultimo carattere di **a** e *il resto* di **a** è esso stesso palindromo.
- ▶ Una volta osservata questa cosa, costruire una funzione ricorsiva per il controllo di palindromia è relativamente semplice:

```
def isPal(a):  
    if len(a)==0:  
        return True  
    else  
        return (a[0]==a[-1]) and isPal(a[1:-1])
```

## Sezione 5

### Moduli e File

# Moduli — I

- ▶ Oltre alla modalità *shell*, sappiamo che è possibile interagire con l'interprete **Python** attraverso i *file*.
  - ▶ Tutte le dichiarazioni di funzione e tutte le istruzioni del vostro programma finiranno nello stesso file.
  - ▶ I file **Python** hanno di solito estensione **py** (ad esempio `fact.py`, `fib.py`, etc.)
- ▶ Quando il programma diventa troppo grande e troppo complesso, conviene ricorrere ai **moduli**.
  - ▶ Un modulo non è nient'altro che un normale programma **Python** che (normalmente) contiene molte dichiarazioni di variabili e funzioni.
  - ▶ È possibile “incorporare” tutte le definizioni di un modulo in un altro programma (o modulo) tramite l'istruzione **import**.
- ▶ I moduli risultano oltremodo utili quando si tratti di rendere il proprio codice *disponibile* alla comunità dei programmatori **Python**.
  - ▶ Basterà, ad esempio, mettere in rete il proprio file **py**, opportunamente documentato.

## Moduli — II

- ▶ Ad esempio, potremmo avere il seguente modulo `cerchio.py`

```
pi = 3.14159
def area(raggio):
    return pi*(raggio**2)
def circonferenza(raggio):
    return 2*pi*raggio
```

- ▶ A questo punto è possibile accedere a tutte le funzionalità attraverso la parola chiave `import`:

```
>>> import cerchio
>>> cerchio.pi
3.14159
>>> cerchio.area(3)
28.27431
>>> cerchio.circonferenza(4)
25.13272
```

## Moduli — III

- ▶ Funzioni e variabili di un modulo *devono* essere accedute attraverso la sintassi  $\langle modulo \rangle . \langle variabile \rangle$  oppure  $\langle modulo \rangle . \langle funzione \rangle$ .
- ▶ Un modo per evitare di dover utilizzare questa sintassi un po' prolissa consiste nell'uso dell'istruzione **from**:

```
>>> from cerchio import *  
>>> pi  
3.14159
```

- ▶ Le variabili di un modulo possono essere **modified**, e in questo modo influenzare il comportamento delle funzioni.  
Ad **esempio**:

```
>>> import cerchio  
>>> cerchio.pi  
3.14159  
>>> cerchio.pi=3.14159265  
>>> cerchio.pi  
3.14159265  
>>> cerchio.area(3)  
28.27433385
```



## File — I

- ▶ Finora, l'unico modo che hanno i nostri programmi **Python** per interagire con i dispositivi di I/O sono l'istruzione **print** e la funzione **raw\_input**.
- ▶ Spesso risulta importante avere la possibilità di manipolare uno o più *file* da un programma:
  - ▶ Per esempio, potremmo voler *leggere* dei dati da un file di testo o *scrivere* delle informazioni su un altro file.
- ▶ Il modo che **Python** mette a disposizione per gestire i file è il cosiddetto **file handle**, che non è altro che un'astrazione del concetto di *file*.
- ▶ Un file handle **Python** supporta tre tipi di operazioni:
  - ▶ L'**apertura**, con cui il file handle viene messo in relazione con un file fisico.
  - ▶ Le **operazioni di modifica**, come la scrittura di una linea e la scrittura di una linea.
  - ▶ La **chiusura**, con cui si dichiara che il file handle non è più necessario.

## File — II

### ► **Apertura.**

- L'*apertura* di un file è eseguita attraverso la funzione `open`, che prende come parametri il nome del file (una stringa) e la modalità di apertura (un'altra stringa, nel nostro caso `'w'` oppure `'r'`).
- Nella modalità `'r'`, il file viene aperto in *lettura*, nella modalità `'w'` invece viene aperto in *scrittura*.

### ► **Modifica.**

- La *lettura* da un file può essere effettuata attraverso un ciclo `for`, in cui il file handle è la sequenza (di stringhe) su cui iterare.
- La *scrittura* su un file può invece essere eseguita attraverso il *metodo* `write`, che prende in input la stringa da scrivere nel file.

### ► **Chiusura.**

- La *chiusura* viene effettuata tramite il metodo `close` che non prende parametri.
- Le operazioni di apertura, modifica e chiusura vanno sempre eseguite in questo preciso ordine.

► **Esempio:**

```
primohandle=open('file1.txt','w')
a=raw_input('Inserisci il tuo nome: ')
primohandle.write(a)
primohandle.close()
```

► **Esempio:**

```
secondohandle=open('file2.txt','r')
for e in secondohandle:
    print e
secondohandle.close()
```

- Esiste anche una terza modalità d'accesso ai file, ossia la modalità 'a', che si comporta come la modalità 'w', ma con la differenza che il vecchio contenuto del file non viene cancellato.

## Sezione 6

### Tipi Strutturati

## Tuple — I

- ▶ Una **tupla** Python non è nient'altro che una sequenza ordinata di elementi, esattamente come una stringa.
- ▶ A differenza delle stringhe, però, gli elementi di una tupla possono essere *di qualunque* tipo.
- ▶ I letterali di tipo tupla sono delimitati dalle parentesi tonde ( e ).
- ▶ Esattamente come le stringhe, le tuple supportano le seguenti operazioni:
  - ▶ La **concatenazione**, tramite l'operatore +;
  - ▶ L'**indicizzazione**;
  - ▶ Lo **slicing**.
- ▶ Il comando **for** può essere utilizzato per iterare sugli elementi di una tupla.
  - ▶ In altre parole, le tuple sono, come le stringhe, un particolare tipo di **sequenza**.
- ▶ Le tuple possono essere visualizzate a video tramite il comando **print**.

## Tuple — II

```
>>> t = (12,4.5,True)
>>> s = (t,34,t)
>>> print t
(12, 4.5, True)
>>> print s
((12, 4.5, True), 34, (12, 4.5, True))
>>> r = s[0:2]
>>> print r
((12, 4.5, True), 34)
>>> print (t+s+s)[4]
34
>>> print (s+t+t)[1:-1]
(34, (12, 4.5, True), 12, 4.5, True, 12, 4.5)
>>> u = (1)
>>> v = (1,)
>>> print u, v
1 (1,)
```

## Tuple — III

- ▶ Le tuple sono particolarmente efficaci in tutte le situazioni in cui il risultato di una funzione sia il risultato di una ricerca di tipo esaustivo.
- ▶ Ad **esempio**, ci si potrebbe chiedere quali siano tutti i divisori comuni di due numeri interi positivi  $n$  e  $m$ . La seguente funzione Python risolve proprio questo problema:

```
def trovaDivisori(n,m):  
    divisori=()  
    for i in range(1,min(n,m)+1):  
        if (n%i==0) and (m%i==0):  
            divisori = divisori + (i,)   
    return divisori
```

## Liste — I

- ▶ Le **liste** risultano in tutto e per tutto simili alle tuple, e supportano come le tuple le operazioni di concatenazione, indicizzazione, e slicing.
- ▶ Ciò che contraddistingue le liste dalle tuple sono le parentesi da usare nella costruzione dei letterali, che nelle liste sono le parentesi quadre ([ e ]).

- ▶ **Esempi:**

```
>>> a=[1,True,[5,[7,6.5]]]  
>>> a[1:3][1][1]  
[7, 6.5]  
>>> a[1:3][1][1][1]  
6.5  
>>> (a+a)[4]  
True
```

- ▶ A differenza delle tuple (e degli oggetti di tutti gli altri tipi che conosciamo), le liste sono **oggetti mutabili**

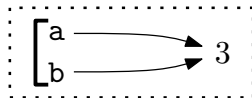
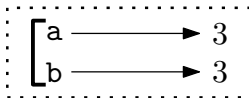


# Oggetti Mutabili e Immutabili — I

- Dopo l'esecuzione di un semplicissimo programma Python come il seguente

```
>>> a=3  
>>> b=a
```

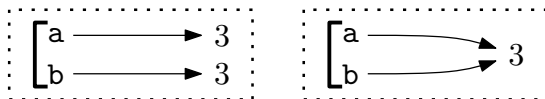
ci aspetteremmo di avere a che fare con il diagramma di stato a sinistra. In realtà la situazione è più simile al diagramma di stato di destra!



- Ciò può essere verificato come segue:

```
>>> id(a)  
140452296722632  
>>> id(b)  
140452296722632
```

## Oggetti Mutabili e Immutabili — II



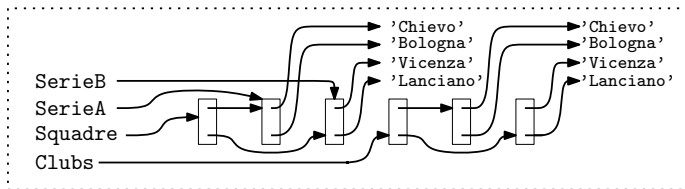
- ▶ Confondere i due diagrammi di stato di cui sopra, però, non crea nessun problema, proprio perchè gli oggetti di tipo `int` sono **immutabili**.
- ▶ Una volta creati, gli oggetti immutabili non possono essere *modificati*. L'unico modo per alterarne il valore è *crearne di nuovi*.
- ▶ Le liste sono invece oggetti **mutabili**.
  - ▶ Esiste la possibilità di alterare una lista senza distruggerla.
  - ▶ Lo stesso oggetto è quindi riferito da variabili distinte, e si parla in tal caso di **aliasing**.

## Liste — II

- Supponiamo di voler rappresentare tramite delle liste alcune squadre di calcio del Campionato Italiano.
- Potremmo procedere come segue:

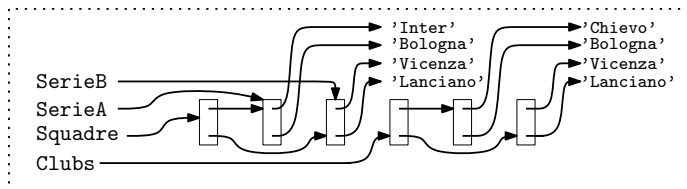
```
SerieA=[ 'Chievo', 'Bologna' ]  
SerieB=[ 'Vicenza', 'Lanciano' ]  
Squadre=[SerieA, SerieB]  
Clubs=[[ 'Chievo', 'Bologna' ], [ 'Vicenza', 'Lanciano' ]]
```

- Dopo questi assegnamenti, il diagramma di stato si presenterebbe simile al seguente:



## Liste — III

- Se eseguiamo l'assegnamento `SerieA[0]='Inter'` il diagramma di stato diventerà:



- Di conseguenza:

```
>>> print Squadre
[['Inter', 'Bologna'], ['Vicenza', 'Lanciano']]
>>> print Clubs
[['Chievo', 'Bologna'], ['Vicenza', 'Lanciano']]
```

## Liste — Alcune Istruzioni Utili

```
>>> x=[1,2,6]
>>> x[2]=3
>>> x.append(4)
>>> x.count(3)
1
>>> x.insert(4,5)
>>> x
[1, 2, 3, 4, 5]
>>> x.remove(3)
>>> x
[1, 2, 4, 5]
>>> x.reverse()
>>> x
[5, 4, 2, 1]
>>> x.sort()
>>> x
[1, 2, 4, 5]
```

## Liste — Cloning

- ▶ Se **x** è una lista, l'assegnamento **y=x** non crea **una nuova** lista, ma fa in modo che **y** punti alla stessa lista cui punta **x**.
- ▶ E' possibile creare una **copia** di una lista esistente?
- ▶ La risposta è affermativa: basta utilizzare l'operatore **[:]**, che è nient'altro che una variante dell'operatore di slicing.
- ▶ **Esempio:**

```
>>> x=[1,2,3]
>>> y=x
>>> y[2]=7
>>> x
[1, 2, 7]
>>> z=x[:]
>>> z[0]=4
>>> x
[1, 2, 7]
>>> z
[4, 2, 7]
```