Un'introduzione a Python

7 maggio 2012

Programmi e linguaggi

Programma: Serie di istruzioni che indicano al computer quali operazioni svolgere.

Programmi e linguaggi

Programma: Serie di istruzioni che indicano al computer quali operazioni svolgere.

Linguaggi di programmazione

- Basso livello
- **Alto livello** ⟵ Python

- Compilati
- **Interpretati** ⟵ Python

Programmi e linguaggi

Programma: Serie di istruzioni che indicano al computer quali operazioni svolgere.

Linguaggi di programmazione

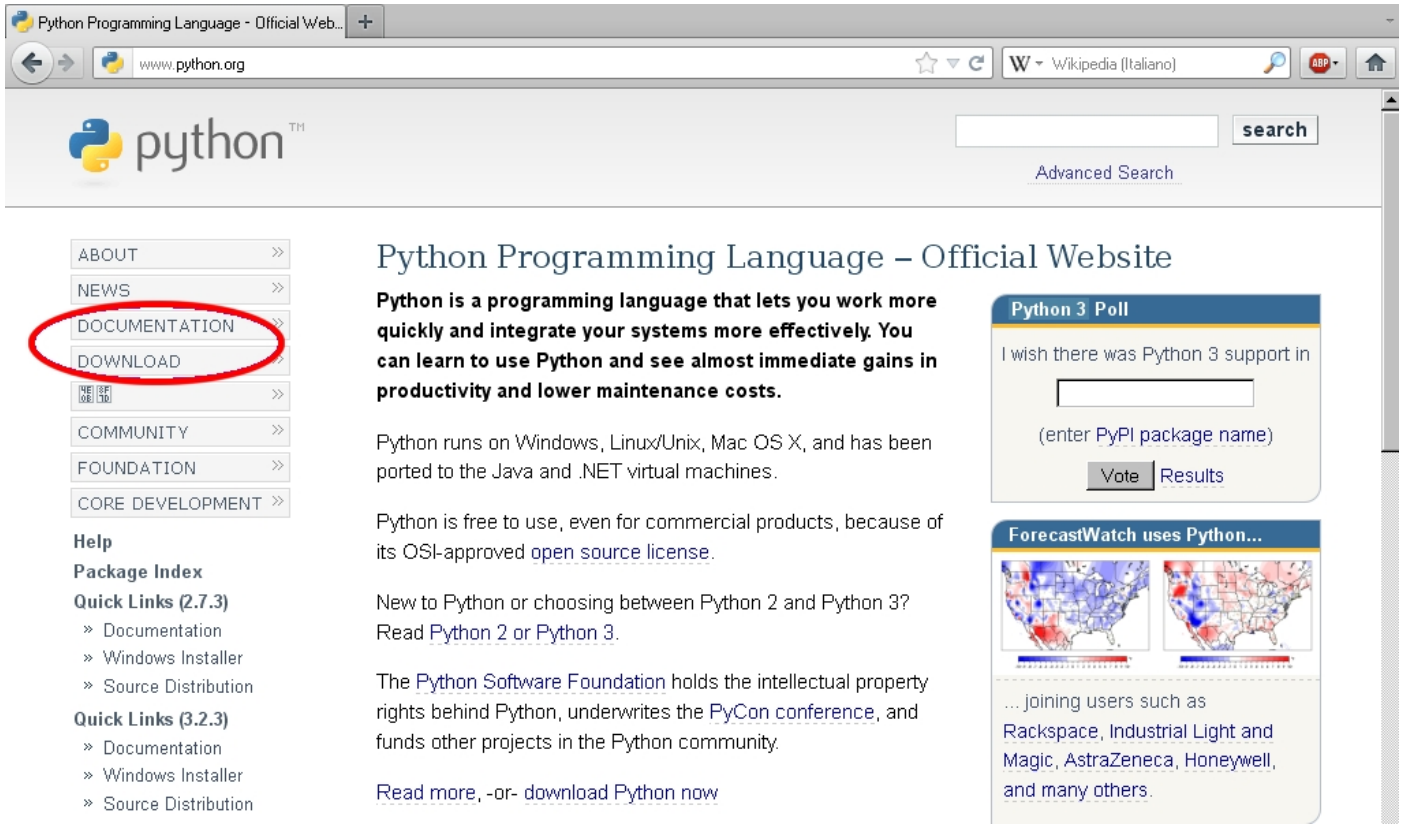
- Basso livello
- **Alto livello** ⟵ Python

- Compilati
- **Interpretati** ⟵ Python

Python è un linguaggio **orientato agli oggetti**

Come ottenere Python

www.python.org



Un primo programma

Avviare l'interprete Python, digitare

```
print 'Ciao mondo!'
```

ed alla fine premere Invio.

Un primo programma

Avviare l'interprete Python, digitare

```
print 'Ciao mondo!'
```

ed alla fine premere Invio.

Il comando `print` visualizza a schermo la **stringa** 'Ciao mondo!'.

Un primo programma

Avviare l'interprete Python, digitare

```
print 'Ciao mondo!'
```

ed alla fine premere Invio.

Il comando `print` visualizza a schermo la **stringa** 'Ciao mondo!'.

Stringa: Collezione ordinata di caratteri compresi tra apici ', oppure doppi apici " oppure tripli apici """

Operare sulle stringhe

I singoli caratteri possono essere essere recuperati dalla stringa nel modo seguente:

```
stringa = ``Stringa di prova''
a = stringa[0]
b = stringa[3]

# ad a viene assegnato il primo carattere di stringa
# a b viene assegnato il quarto carattere di stringa
```

Operare sulle stringhe

I singoli caratteri possono essere essere recuperati dalla stringa nel modo seguente:

```
stringa = ``Stringa di prova''
a = stringa[0]
b = stringa[3]

# ad a viene assegnato il primo carattere di stringa
# a b viene assegnato il quarto carattere di stringa
```

Attenzione: gli indici partono dallo zero!
Attenzione: le stringhe non sono modificabili!

```
stringa[2] = 'c'
```

produce un errore.

Ricerca nelle strighe

Obiettivo: Data una stringa `text` ed un carattere `pattern`, verificare se `pattern` sia presente o meno in `text`.

```
# definizione della stringa e del carattere

text = "Oggi e' il 7 maggio"
pattern = "g"
```

Ricerca nelle strighe

Obiettivo: Data una stringa `text` ed un carattere `pattern`, verificare se `pattern` sia presente o meno in `text`.

```
# definizione della stringa e del carattere

text = "Oggi e' il 7 maggio"
pattern = "g"

# utilizzo dell'operatore in

pattern in text

# la precedente istruzione restituisce True
```

Migliorare l'output

```
# definizione della stringa e del carattere

text = "Oggi e' il 7 maggio"
pattern = "g"

# costrutto if ... else ...

if pattern in text:
    print "C'e'!"
else:
    print "Non c'e'!"
```

Il costrutto `if...else...`

Il costrutto `if...else...` viene utilizzato per prendere delle **decisioni**.
La sua *sintassi* è la seguente:

```
if condizione :
    istruzione1
    istruzione2
else:
    istruzione3
    istruzione4
```

Attenzione: i due punti : !

Attenzione: indentazione!

Il costrutto `if...else...`

Il costrutto `if...else...` viene utilizzato per prendere delle **decisioni**.
La sua *sintassi* è la seguente:

```
if condizione :
    istruzione1
    istruzione2
else:
    istruzione3
    istruzione4
```

Attenzione: i due punti : !

Attenzione: indentazione!

Funzionamento:
Innanzitutto `if` valuta la condizione:

- se essa è vera, allora esegue istruzione1 ed istruzione2
- se essa è falsa, allora esegue istruzione3 ed istruzione4

Migliorare l'input

Come gestire stringhe e caratteri inseriti dall'utente?
Si utilizza il comando `raw_input()`

Migliorare l'input

Come gestire stringhe e caratteri inseriti dall'utente?
Si utilizza il comando `raw_input()`

```
# il programma richiede i due dati

text = raw_input("Inserisci la stringa: ")
pattern = raw_input("Inserisci il carattere: ")

if pattern in text:
    print "C'e'!"
else:
    print "Non c'e'!"
```

Migliorare la riutilizzabilità

Obiettivo: fare in modo che sia possibile "richiamare" il codice finora scritto un numero arbitrario di volte, con input (possibilmente) differenti.

Migliorare la riutilizzabilità

Obiettivo: fare in modo che sia possibile "richiamare" il codice finora scritto un numero arbitrario di volte, con input (possibilmente) differenti.

Si definisce una **funzione**.

```
def cerca(text, pattern):
    if pattern in text:
        print "C'e'!"
    else:
        print "Non c'e'!"
```

Funzioni in Python

La *sintassi* di una funzione in Python è la seguente:

```
def nome_della_funzione(argomenti) :
    istruzione1
    istruzione2
```

Attenzione: i due punti : !
Attenzione: indentazione!

Una funzione viene richiamata in questo modo:

```
nome_della_funzione(argomenti_passati_alla_funzione)
```

Tipi di dati

In Python esistono vari tipi di dato:

- Stringhe
- Numeri
- Liste
- Tuple
- Dizionari

Tipi di dati

In Python esistono vari tipi di dato:

- Stringhe
- Numeri
- Liste
- Tuple
- Dizionari

I tipi dato possono essere manipolati attraverso degli **operatori**:

```
'Una ' + 'concatenazione ' + ' di stringhe'
(7*12) / 6
```

Numeri

Osserviamo cosa succede in questo caso:

```
12 / 5

# l'output e' 2
```

Numeri

Osserviamo cosa succede in questo caso:

```
12 / 5

# l'output e' 2
```

Il risultato 'imprevisto' è dovuto al fatto che Python implementa quattro tipi differenti di numeri:

- `int` (interi con segno)
- `long` (interi "lunghi" con segno)
- `float` (numeri reali a virgola mobile)
- `complex` (numeri complessi, l'unità immaginaria è `j`)

Numeri

Il risultato "corretto" si ha effettuando un'operazione di **cast**:

```
float(12)/5
```

Utilizzare l'operatore di cast `float()` provoca i seguenti risultati:

- il numero 12 viene "pensato" come un numero a virgola mobile;
- il numero 5 viene "promosso" a virgola mobile;
- l'operazione si svolge all'interno del tipo dato `float`.

In questo caso l'output è `2.4`.

Liste

Uno dei tipi dati più versatili in Python è la **lista**. Le liste sono "contenitori" ordinati di variabili di tipo (possibilmente) differente. Ad esempio:

```
lista = ['mela', 'fragola', 12, 17]
```

Liste

Uno dei tipi dati più versatili in Python è la **lista**. Le liste sono "contenitori" ordinati di variabili di tipo (possibilmente) differente. Ad esempio:

```
lista = ['mela', 'fragola', 12, 17]
```

Si ha accesso agli elementi di una lista come per le stringhe

```
a = lista[1]
```

Liste

Uno dei tipi dati più versatili in Python è la **lista**. Le liste sono "contenitori" ordinati di variabili di tipo (possibilmente) differente. Ad esempio:

```
lista = ['mela', 'fragola', 12, 17]
```

Si ha accesso agli elementi di una lista come per le stringhe

```
a = lista[1]
```

Le liste possono contenere a loro volte altre liste (si parla in questo caso di **liste annidate**):

```
lista = [[1,2,3], [4,5,6], [7,8,9]]
```

Osservazione: attraverso le liste annidate è possibile rappresentare le matrici.

Operare con le liste

Attenzione: le liste sono modificabili!
In Python vi sono molti **metodi** predefiniti per lavorare con le liste:

- `lista.append(x)`
  Aggiunge l'elemento `x` alla lista.
- `lista.extend(L)`
  Estende la lista data concatenandovi la lista `L`.
- `lista.insert(i,x)`
  Inserisce l'elemento `x` alla lista in posizione `i`.
- `lista.pop(i)`
  Rimuove dalla lista l'elemento in posizione `i` e restituisce il suo valore.

Scorrere le liste

Obiettivo: Data una lista di numeri, visualizzare a schermo ciascuno di essi, dopo avergli sommato 1.
Per ottenere questo risultato facciamo uso di un **ciclo di iterazione**.

Scorrere le liste

Obiettivo: Data una lista di numeri, visualizzare a schermo ciascuno di essi, dopo avergli sommato 1.
Per ottenere questo risultato facciamo uso di un **ciclo di iterazione**.

```
# la lista viene inizializzata

lista = [5,2,9,8,4]

# comincia il ciclo di iterazione

for i in lista:
    print i+1
```

Il ciclo `for`

Il ciclo `for` ha la seguente *sintassi*:

```
for variabile in lista :
    istruzione1
    istruzione2
```

Il ciclo `for`

Il ciclo `for` ha la seguente *sintassi*:

```
for variabile in lista :
    istruzione1
    istruzione2
```

Funzionamento:

- La variabile assume di volta in volta il valore di uno degli elementi della lista, dal primo all'ultimo.
- Dopo che alla variabile è stato assegnato il valore di un elemento della lista vengono eseguite istruzione1 ed istruzione2, ed in seguito si passa all'elemento successivo.

Liste di numeri

Il comando `range(inizio,fine,step)` restituisce un tipo particolare di liste di numeri. Esse:

- partono da **`inizio`** (compreso)
- arrivano a **`fine`** (escluso)
- procedono con un passo dato da **`step`**

In questo modo l'istruzione `range(0,10,2)` restituisce la lista `[0,2,4,6,8]`.

Liste di numeri

Il comando `range(inizio,fine,step)` restituisce un tipo particolare di liste di numeri. Esse:

- partono da **`inizio`** (compreso)
- arrivano a **`fine`** (escluso)
- procedono con un passo dato da **`step`**

In questo modo l'istruzione `range(0,10,2)` restituisce la lista `[0,2,4,6,8]`.

Attenzione: se `inizio` e `step` vengono omessi, allora la lista parte da `0` ed ha un passo pari ad `1`.

In questa situazione può rivelarsi molto utile l'istruzione `len(lista)`, che restituisce la lunghezza di una lista data.

Costruire liste

Le liste possono essere costruite:

- descrivendo esplicitamente i loro elementi;
- modificandone di già esistenti con i metodi `append`, `extend`, `insert` e `remove`;
- creandone a partire da liste già esistenti, usando la *notazione a fette*;
- utilizzando le **list comprehensions**.

Costruire liste

Le liste possono essere costruite:

- descrivendo esplicitamente i loro elementi;
- modificandone di già esistenti con i metodi `append`, `extend`, `insert` e `remove`;
- creandone a partire da liste già esistenti, usando la *notazione a fette*;
- utilizzando le **list comprehensions**.

Notazione a fette

Attraverso la notazione a fette è possibile manipolare porzioni di una lista. Data una lista `lista`, allora l'espressione

```
lista[a:b]
```

restituisce una lista della forma seguente:

```
[ lista[a], ..., lista[b-1] ]
```

List comprehensions

Obiettivo: data una lista di numeri, crearne un'altra che contenga i resti modulo 3 dei numeri della lista data.

List comprehensions

Obiettivo: data una lista di numeri, crearne un'altra che contenga i resti modulo 3 dei numeri della lista data.
Un modo possibile è:

```
# inizializza la lista di numeri
lista = [4,7,8,2,5,1]

# inizializza una lista vuota
resti = []

# riempimento della lista dei resti
for x in lista:
    resti.append(x % 3)
```

List comprehensions

Le precedenti istruzioni possono essere rese più concise:

```
# inizializza la lista di numeri
lista = [4,7,8,2,5,1]

# inizializza la lista di resti
resti = [x % 3 for x in lista]
```

List comprehensions

Le precedenti istruzioni possono essere rese più concise:

```
# inizializza la lista di numeri
lista = [4,7,8,2,5,1]

# inizializza la lista di resti
resti = [x % 3 for x in lista]
```

La *sintassi* di una list comprehension è la seguente:

[espressione **for** variabile1 **in** lista1 **for** variabile2 **in** lista2 **if** condizione1 **if** condizione2 **]**

Ancora funzioni

Le funzioni di Python possono richiamare altre funzioni.
Supponiamo di voler scrivere una funzione che calcoli i valori della seguente applicazione:

$$f(x) = \sum_{i=1}^{100} P_i(x) \quad \text{dove} \quad P_i(x) = \frac{x}{i} + i$$

Ancora funzioni

Le funzioni di Python possono richiamare altre funzioni.
Supponiamo di voler scrivere una funzione che calcoli i valori della seguente applicazione:

$$f(x) = \sum_{i=1}^{100} P_i(x) \quad \text{dove} \quad P_i(x) = \frac{x}{i} + i$$

```
# definizione della funzione che calcola P_i(x)
def p(i, x):
    return (x/i + i)

# definizione della funzione principale
def f(x):
    temp = 0
    for i in range(1,101):
        temp += p(i,x)
    return temp
```

Ricorsione

Una funzione può anche richiamare sé stessa: si parla in questo caso di **ricorsione**.
Un esempio di ricorsione si ha nel calcolo del fattoriale:

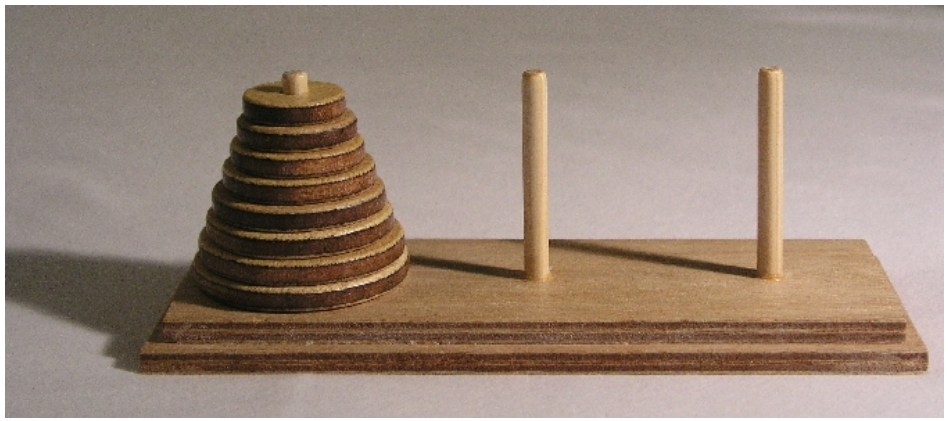
```
def fattoriale(n):
    if n == 1:
        temp = 1
    else:
        temp = n * fattoriale(n-1)
    return temp
```

Attenzione: l'uso della ricorsione ha un costo computazionale molto alto!

La Torre di Hanoi

Nel 1883 Eduard Lucas, matematico francese, inventa un gioco che chiama *La Torre di Hanoi*.

(a) Eduard Lucas



(b) Torre di Hanoi

Uso dello pseudocodice

Obiettivo: listare le mosse per risolvere il problema della Torre di Hanoi

Dal momento che questo programma è più complesso di quelli visti in precedenza, ci serviremo di uno strumento di sviluppo, lo *pseudocodice*.

Uso dello pseudocodice

Obiettivo: listare le mosse per risolvere il problema della Torre di Hanoi

Dal momento che questo programma è più complesso di quelli visti in precedenza, ci serviremo di uno strumento di sviluppo, lo *pseudocodice*.

Funzione che sposta i primi (dall'alto) `numero` dischi dall'asta `partenza` all'asta `arrivo` usando come "sponda" l'asta `scambio`

```
Se numero e' uguale ad 1 allora
    Sposta il primo disco da partenza ad arrivo
Altrimenti
    Sposta i primi (numero - 1) dischi da partenza
    a scambio, usando arrivo come sponda;
    Sposta il primo disco da partenza ad arrivo;
    Sposta i primi (numero - 1) dischi da scambio
    ad arrivo, usando partenza come sponda
```

Il ciclo `while`

Accanto al ciclo `for`, tra i cicli di iterazione di Python è presente il ciclo `while`. La sua *sintassi* è la seguente:

```
while condizione :
    istruzione1
    istruzione2
```

Funzionamento

Il ciclo `while` si ripete (ovvero, vengono eseguite istruzione1 ed istruzione2) fino a quando la condizione rimane vera.

Attenzione: le istruzioni che sono inserite all'interno del ciclo `while` devono prima o poi rendere falsa la condizione!

Conteggi

Obiettivo: creare una lista di 100 numeri casuali e verificare che sono distribuiti in modo equiprobabile.

Conteggi

Obiettivo: creare una lista di 100 numeri casuali e verificare che sono distribuiti in modo equiprobabile.

Per avere a disposizione numeri casuali useremo il metodo **`random`** del modulo **`random`**. Esso si richiama nel modo seguente:

```
import random

random.random()
```

L'ultima istruzione restituisce un numero casuale tra 0.0 (compreso) e 1.0 (escluso).

Conteggi

Obiettivo: creare una lista di 100 numeri casuali e verificare che sono distribuiti in modo equiprobabile.

Per avere a disposizione numeri casuali useremo il metodo **`random`** del modulo **`random`**. Esso si richiama nel modo seguente:

```
import random

random.random()
```

L'ultima istruzione restituisce un numero casuale tra 0.0 (compreso) e 1.0 (escluso).

Una volta costruita la lista, dobbiamo dividere l'intervallo $[0, 1)$ in un numero fissato di sottointervalli.

Conteggi

A questo punto abbiamo bisogno di una funzione che scorra la lista e verifichi se ci sono elementi sono in un determinato sottointervallo, ed in tal caso aumenti il rispettivo indice.

Conteggi

A questo punto abbiamo bisogno di una funzione che scorra la lista e verifichi se ci sono elementi sono in un determinato sottointervallo, ed in tal caso aumenti il rispettivo indice.

Osservazione: questo algoritmo è molto inefficiente, perché per ogni sottointervallo viene percorsa l'intera lista di numeri casuali.

Osservazione: se N è la lunghezza della lista ed M è il numero di sottointervalli, allora il tempo di computazione di questo algoritmo è proporzionale a $N \cdot M$.

Conteggi

A questo punto abbiamo bisogno di una funzione che scorra la lista e verifichi se ci sono elementi sono in un determinato sottointervallo, ed in tal caso aumenti il rispettivo indice.

Osservazione: questo algoritmo è molto inefficiente, perché per ogni sottointervallo viene percorsa l'intera lista di numeri casuali.

Osservazione: se N è la lunghezza della lista ed M è il numero di sottointervalli, allora il tempo di computazione di questo algoritmo è proporzionale a $N \cdot M$.

È possibile modificare l'algoritmo in modo che il tempo di computazione non dipenda dal numero di sottointervalli?

Conteggi

Costruiamo la lista di numeri casuali:

```
import random
lista = [random.random() for i in range(100)]
```

Dividiamo ora $[0, 1)$ in 8 intervalli:

Conteggi

Costruiamo la lista di numeri casuali:

```
import random
lista = [random.random() for i in range(100)]
```

Dividiamo ora $[0, 1)$ in 8 intervalli:

```
numero_intervalli = 8
ampiezza_intervallo = 1.0 / 8

# creazione di una lista con
# gli estremi di ogni sottointervallo

intervalli = []
for i in range(numero_intervalli):
    intervalli.append([i*ampiezza_intervallo, \
                       (i+1)*ampiezza_intervallo])
```

Conteggi

Ora definiamo una funzione che effettui il conteggio:

```
def conta(lista, estremi):
    contatore = 0
    for x in lista:
        if estremi[0] <= x < estremi[1]:
         contatore += 1
    return contatore
```

Conteggi

Ora definiamo una funzione che effettui il conteggio:

```
def conta(lista, estremi):
    contatore = 0
    for x in lista:
        if estremi[0] <= x < estremi[1]:
         contatore += 1
    return contatore
```

A questo punto nel corpo centrale del programma la funzione `conta` dovrà essere richiamata passandole di volta in volta un elemento della lista `intervalli` ed incrementando un opportuno contatore.

Conteggi

Ora definiamo una funzione che effettui il conteggio:

```
def conta(lista, estremi):
    contatore = 0
    for x in lista:
        if estremi[0] <= x < estremi[1]:
         contatore += 1
    return contatore
```

A questo punto nel corpo centrale del programma la funzione `conta` dovrà essere richiamata passandole di volta in volta un elemento della lista `intervalli` ed incrementando un opportuno contatore.

Osservazione: in questo modo la funzione `conta` viene richiamata `numero_intervalli` volte.

Conteggi

È possibile migliorare questo algoritmo?
Notiamo che nella funzione `conta` per stabilire se l'elemento x della lista si trova nell'i-esimo intervallo si effettua il confronto seguente:

$$i \cdot \texttt{ampiezza\_intervallo} \leq x < (i+1) \cdot \texttt{ampiezza\_intervallo}$$

Conteggi

È possibile migliorare questo algoritmo?
Notiamo che nella funzione `conta` per stabilire se l'elemento x della lista si trova nell'i-esimo intervallo si effettua il confronto seguente:

$$i \cdot \texttt{ampiezza\_intervallo} \leq x < (i+1) \cdot \texttt{ampiezza\_intervallo}$$

Ricordando la definizione di `ampiezza_intervallo` abbiamo:

$$\frac{i}{\texttt{numero\_intervalli}} \leq x < \frac{i+1}{\texttt{numero\_intervalli}}$$

Conteggi

È possibile migliorare questo algoritmo?
Notiamo che nella funzione `conta` per stabilire se l'elemento x della lista si trova nell'i-esimo intervallo si effettua il confronto seguente:

$$i \cdot \texttt{ampiezza\_intervallo} \leq x < (i+1) \cdot \texttt{ampiezza\_intervallo}$$

Ricordando la definizione di `ampiezza_intervallo` abbiamo:

$$\frac{i}{\texttt{numero\_intervalli}} \leq x < \frac{i+1}{\texttt{numero\_intervalli}}$$

che è equivalente a

$$i \leq \texttt{numero\_intervalli} \cdot x < i+1$$

Conteggi

Dalle ultime disuguaglianze notiamo che

$$i = \underbrace{\lfloor \texttt{numero\_intervalli} \cdot x \rfloor}_{\text{parte intera inferiore}}$$

ovvero la quantità a destra restituisce l'indice dell'intervallo all'interno del quale è compreso l'elemento x.

Conteggi

Dalle ultime disuguaglianze notiamo che

$$i = \underbrace{\lfloor \texttt{numero\_intervalli} \cdot x \rfloor}_{\text{parte intera inferiore}}$$

ovvero la quantità a destra restituisce l'indice dell'intervallo all'interno del quale è compreso l'elemento x.

In questo modo è possibile scrivere un algoritmo che legga una volta sola la lista ed aggiorni tutti i contatori:

```
for x in lista:
    indice = int(numero_intervalli * x
    contatore[indice] +=1
```

Punto della situazione

Fino ad ora abbiamo esaminato i seguenti concetti:

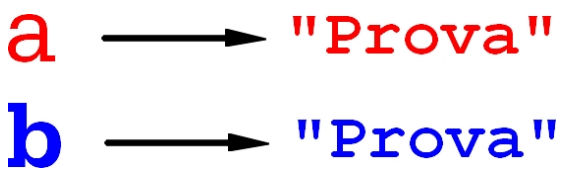
- Tipi di dato
  - Stringhe
  - Numeri
  - Liste
- Strutture di controllo
  - Costrutto `if...else`
- Cicli di iterazione
  - Ciclo `for`
  - Ciclo `while`

Valori e riferimenti

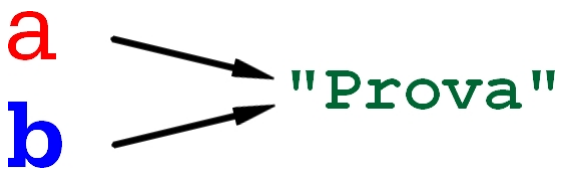
Consideriamo le seguenti due istruzioni:

```
a = ``Prova''
b = a
```

In quale delle seguenti due situazioni ci troviamo?



(a) Le due variabili "puntano" ad indirizzi diversi di memoria



(b) Le due variabili "puntano" al medesimo indirizzo di memoria

Lo si può controllare attraverso il comando

```
id(a) == id(b)
```

Valori e riferimenti

Questo fenomeno accade anche con le liste, dunque

```
a = [1,2,3,4]
b = a

a[2] = [0]
```

modifica sia `a` che `b`.

Valori e riferimenti

Questo fenomeno accade anche con le liste, dunque

```
a = [1,2,3,4]
b = a

a[2] = [0]
```

modifica sia `a` che `b`.
Per ovviare a questo problema si può utilizzare la notazione a fette:

```
a = [1,2,3,4]
b = a[:]

a[2] = [0]

print a # l'output e' [1,2,0,4]
print b # l'output e' [1,2,3,4]
```

Valori e riferimenti

Purtroppo la notazione a fette non funziona nel caso delle liste annidate. Per effettuare una copia di una lista annidata bisogna utilizzare il **metodo** `deepcopy()` presente nel **modulo** `copy`, che viene richiamato in questo modo:

```
import copy

a = [[1,2],[3,4]]
b = copy.deepcopy(a)
```

Valori e riferimenti

Purtroppo la notazione a fette non funziona nel caso delle liste annidate. Per effettuare una copia di una lista annidata bisogna utilizzare il **metodo** `deepcopy()` presente nel **modulo** `copy`, che viene richiamato in questo modo:

```
import copy

a = [[1,2],[3,4]]
b = copy.deepcopy(a)
```

Una situazione differente si ha in questo caso:

```
stringa1 = ''Zucchero''
stringa2 = ''Zucchero''

lista1 = [1,2,3,4]
lista2 = [1,2,3,4]
```

Argomenti opzionali

Alle funzioni possono essere passati argomenti opzionali:

```
def funzione(numero, citta = 'Roma', abitanti = 2000000)
    print 'Hai inserito il numero ', numero
    print '\n'
    print citta, ''' e' una citta' italiana'''
    print 'con ', abitanti, ' abitanti'
```

Argomenti opzionali

Alle funzioni possono essere passati argomenti opzionali:

```
def funzione(numero, citta = 'Roma', abitanti = 2000000)
    print 'Hai inserito il numero ', numero
    print '\n'
    print citta, ''' e' una citta' italiana'''
    print 'con ', abitanti, ' abitanti'
```

In questo caso la funzione può essere richiamata in vari modi:

```
funzione(5)

funzione(10, 'Trieste', 200000)

funzione(15, abitanti = 1000000)
```

Istruzioni `break` e `continue`

Le istruzioni `break` e `continue` possono essere usate per interrompere una o più iterazioni di un ciclo `for` o `while`.

Istruzioni `break` e `continue`

Le istruzioni `break` e `continue` possono essere usate per interrompere una o più iterazioni di un ciclo `for` o `while`.

Si consideri l'esempio seguente:

```
termine = input(``Inserire termine: ``)

for i in range(100):
    ...
    if i == termine:
        break
    ...
```

L'istruzione `break` fa terminare il ciclo `for`, ignorando tutte le iterazioni successive a quella in cui essa si presenta.

Istruzioni `break` e `continue`

A differenza dell'istruzione `break`, l'istruzione `continue` permette di ignorare le istruzioni successive ad essa relative alla presente iterazione, facendo ricomiciare il ciclo dall'iterazione successiva.

Istruzioni `break` e `continue`

A differenza dell'istruzione `break`, l'istruzione `continue` permette di ignorare le istruzioni successive ad essa relative alla presente iterazione, facendo ricomiciare il ciclo dall'iterazione successiva.

Si considera l'esempio seguente:

Tuple

Una **tupla** è una collezione ordinata di elementi, simile ad una lista.
A differenza delle liste, le tuple non sono modificabili.

```
coordinate = (3,4)
```

Tuple

Una **tupla** è una collezione ordinata di elementi, simile ad una lista. A differenza delle liste, le tuple non sono modificabili.

```
coordinate = (3,4)
```

Attraverso le tuple è possibile scambiare due variabili in maniera efficiente. Un modo "classico" per farlo è utilizzare una terza variabile temporanea:

```
b = temp
b = a
a = temp
```

Tuple

Una **tupla** è una collezione ordinata di elementi, simile ad una lista. A differenza delle liste, le tuple non sono modificabili.

```
coordinate = (3,4)
```

Attraverso le tuple è possibile scambiare due variabili in maniera efficiente. Un modo "classico" per farlo è utilizzare una terza variabile temporanea:

```
b = temp
b = a
a = temp
```

In Python ciò si può rendere nel modo seguente:

```
b,a = a,b
```

Dizionari

Un **dizionario** è una collezione non ordinata di elementi. Ciascuno degli elementi viene individuato da una particolare *chiave*:

```
dizionario = {'uno':'one', 'due':'two', 'tre':'three'}
```

Dizionari

Un **dizionario** è una collezione non ordinata di elementi. Ciascuno degli elementi viene individuato da una particolare *chiave*:

```
dizionario = {'uno':'one', 'due':'two', 'tre':'three'}
```

Attenzione: i dizionari sono oggetti modificabili!

Dizionari

Un **dizionario** è una collezione non ordinata di elementi. Ciascuno degli elementi viene individuato da una particolare *chiave*:

```
dizionario = {'uno':'one', 'due':'two', 'tre':'three'}
```

Attenzione: i dizionari sono oggetti modificabili!

Per operare con i dizionari si possono utilizzare i seguenti metodi:

- `dizionario.keys()`
  Restituisce una lista che ha come elementi le chiavi del dizionario
- `dizionario.values()`
  Restituisce una lista che ha come elementi i valori del registro
- `dizionario.items()`
  Restituisce una lista di coppie `(chiave,valore)`

Ancora liste

Python dispone di due funzioni per operare con le liste: `filter()` e `map()`.
La loro sintassi è la seguente:

Ancora liste

Python dispone di due funzioni per operare con le liste: `filter()` e `map()`.
La loro sintassi è la seguente:

`filter``(`funzione`,` lista`)`

dove *funzione* è una funzione che ha un unico argomento e restituisce `True` o `False`.

Ancora liste

Python dispone di due funzioni per operare con le liste: `filter()` e `map()`.
La loro sintassi è la seguente:

`filter`(funzione, lista)

dove *funzione* è una funzione che ha un unico argomento e restituisce `True` o `False`.

Funzionamento: `filter()` restituisce una lista formata dagli elementi della lista in input che, posti come argomenti di *funzione*, restituiscono `True`.

Ancora liste

Python dispone di due funzioni per operare con le liste: `filter()` e `map()`.
La loro sintassi è la seguente:

`filter`(funzione, lista)

dove *funzione* è una funzione che ha un unico argomento e restituisce `True` o `False`.

Funzionamento: `filter()` restituisce una lista formata dagli elementi della lista in input che, posti come argomenti di *funzione*, restituiscono `True`.

`map`(funzione, lista) dove *funzione* è una funzione che ha un unico argomento.

Ancora liste

Python dispone di due funzioni per operare con le liste: `filter()` e `map()`.
La loro sintassi è la seguente:

`filter``(`funzione`,` lista`)`

dove *funzione* è una funzione che ha un unico argomento e restituisce `True` o `False`.

Funzionamento: `filter()` restituisce una lista formata dagli elementi della lista in input che, posti come argomenti di *funzione*, restituiscono `True`.

`map``(`funzione`,` lista`)` dove *funzione* è una funzione che ha un unico argomento.

Funzionamento: `map()` restituisce una lista formata applicando *funzione* a ciascuno degli elementi di *lista*.