Introduzione al linguaggio Python

Corso di Programmazione I

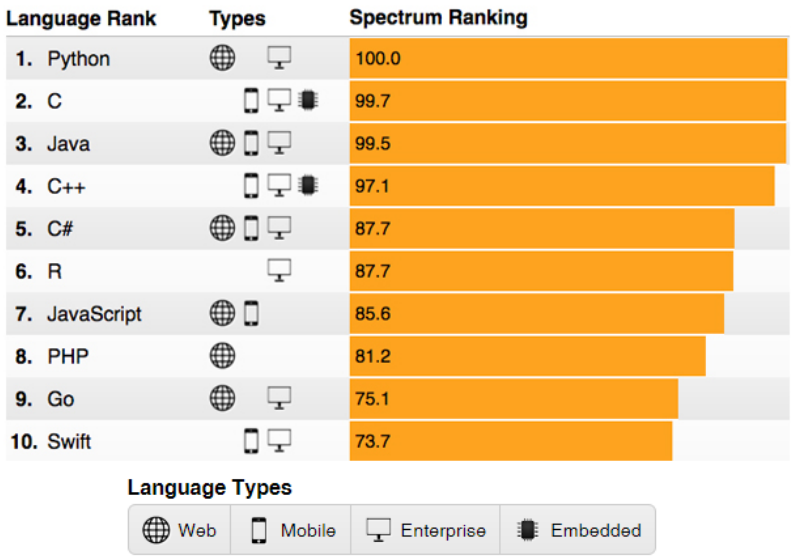
Roberto Canonico Valeria Vittorini

Università degli Studi di Napoli Federico II

A.A. 2017-2018

Obiettivi della lezione

- Introdurre ad un linguaggio di programmazione la cui popolarità sta aumentando negli ultimi anni
- Mostrare come si programma in un linguaggio interpretato
- Spingere ad imparare nuovi linguaggi di programmazione
- Caveat: *saranno mostrati prevalentemente gli aspetti sintattici del linguaggio, tralasciando gli 'internals', cioè i meccanismi utilizzati dall'interprete per eseguire il codice Python ed implementare le astrazioni del linguaggio*



Fonte: IEEE Spectrum, *Interactive: The Top Programming Languages 2017*

Principali caratteristiche del linguaggio Python

Python è un linguaggio di programmazione:

- di alto livello e general-purpose ideato dall'informatico olandese Guido van Rossum all'inizio degli anni novanta
- interpretato
  - interprete open-source multipiattaforma
- con tipizzazione *forte* e *dinamica*
  - il controllo dei tipi viene eseguito a runtime
- supporta il paradigma object-oriented
- con caratteristiche di programmazione funzionale e riflessione
- ampiamente utilizzato per sviluppare applicazioni di scripting, scientifiche, distribuite, e per system testing

Oggi se ne utilizzano prevalentemente due diverse versioni, identificate dai numeri di versione 2.7 e 3.1, e definite entrambe nel 2010.
L'interprete è scaricabile dal web: `https://www.python.org/downloads/`

Interprete Python: modalità interattiva

- Lanciando il programma eseguibile python si esegue l'interprete Python *in modalità interattiva*
- Si ottiene un *prompt* dal quale si possono eseguire singoli *statement* Python
- In modalità interattiva i comandi vanno digitati da tastiera

```
C:\Users\Roberto>python
ActivePython 2.7.8.10 (ActiveState Software Inc.) based on
Python 2.7.8 (default, Jul  2 2014, 19:48:49) [MSC v.1500 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4+7
11
>>> print "Hello world!"
Hello world!
>>>
```

- L'interprete tiene memoria dei comandi eseguiti precedentemente, che possono essere richiamati al prompt usando i tasti freccia-sù e freccia-giù della tastiera

Esecuzione di script Python

- Un programma Python è tipicamente scritto in un file sorgente testuale
  - I file sorgente Python sono anche detti *script*
    perchè possono essere eseguiti direttamente dall'interprete
  - Per convenzione, gli script Python sono salvati in file con estensione `.py`
- L'esecuzione del programma si ottiene avviando l'interprete Python
  e fornendo come input il path dello script Python
  - In Windows:
    ```
    C:\Users\Roberto\Python>python prova.py
    ```
  - In Linux, se il file script è eseguibile:
    ```
    chmod prova.py a+x
    ```
    e la prima riga dello script è:
    ```
    #!/usr/bin/python
    ```
    è possibile eseguire direttamente lo script dal prompt dei comandi:
    ```
    user@computer:~$ prova.py
    ```
- Per lo sviluppo, una pratica comune è quella di usare un normale editor di testo per scrivere il codice sorgente
  - Esistono editor di testo specializzati per la scrittura di programmi, come Notepad++, che evidenziano con colori le keyword del linguaggio

Python: aspetti lessicali

- Un programma Python è una sequenza di linee di testo
- L'unità di esecuzione è detta uno *statement*
- I caratteri di una linea successivi ad un eventuale carattere hash (#) sono considerati un commento ed ignorati dall'interprete
- Una singola linea di testo può contenere più statement separati da punto e virgola

```
a = 5; b = 8; c = 3
```

- Se una riga termina con il carattere backslash (\), l'interprete unisce la riga con la successiva in un'unica *riga logica*

```
# Esempio di statement su due linee di testo
a = 3 + 5 + 7 +\
    4 + 2 + 1
```

- Uno statement termina con la fine di una riga, a meno che la riga non contenga parentesi aperte e non ancora chiuse

```
# Esempio di statement su due linee di testo senza backslash
a = (3 + 5 + 7 +
    4 + 2 + 1)
```

- Il linguaggio usa, per scopi diversi:
  - (e) parentesi tonde - *parentheses*
  - [e] parentesi quadre - *brackets*
  - { e } parentesi graffe - *braces*

Python: indentazione

- Python non usa parentesi per delimitare blocchi di codice
  - A tale scopo, Python usa le regole di indentazione
  - Per *indentazione* si intendono gli spazi (o caratteri di tabulazione) a sinistra del primo carattere dello statement
  - In una sequenza di statement, uno statement deve avere la stessa indentazione del precedente, altrimenti si genera un errore in esecuzione

```
>>> a = 1
>>>    b = 2
  File "<stdin>", line 1
    b = 2
    ^
IndentationError:  unexpected indent
```

- I *compound statement* (es. `if`, `while`, `for`, `try`, `def`, ...) "contengono" una sequenza (*suite* o *body*) di statement elementari
- Il body di un compound statement è formato da linee con la stessa indentazione allineate più a destra rispetto all'istruzione che "le contiene"
  - Python raccomanda di usare 4 caratteri spazio per ciascun livello di indentazione e di non usare il carattere tab

```
def f(a):
    if (a == 5):
       print "numero uguale a 5"
       b = 1
    else:
      print "numero diverso da 5"; b = 0
    print "b = ", b   # Questa istruzione viene eseguita in ogni caso
    return b
```

Lo statement print

- Lo statement `print` stampa una stringa sullo standard output
- Se l'argomento di `print` è un dato di tipo diverso da stringa, `print` ne stampa una rappresentazione testuale del valore
- Se `print` ha più argomenti separati da virgola, i rispettivi valori sono stampati separati da uno spazio

```
>>> print 1, 12, 99
1 12 99
```

- Se la lista di argomenti di `print` termina con una virgola, non viene stampato un ritorno a capo

```
>>> print "Pippo e", ; print "Topolino" ; print "vivono a Topolinia"
Pippo e Topolino
vivono a Topolinia
>>>
```

Sul concetto di oggetto in Python

- Un *oggetto* è un'entità caratterizzata da un insieme di *attributi* e *metodi*
  - Il concetto è ampiamente trattato nel corso, qui lo si assume noto
- In Python "tutto è oggetto"
  - Anche valori di tipi come int, long, float, string sono "oggetti"
  - Esistono attributi e metodi predefiniti associati a questi "tipi"
  - La funzione predefinita `dir()` restituisce una lista di attributi e metodi di un qualunque oggetto

```
>>> dir(-5)
['__abs__', '__add__', '__and__', '__class__', '__cmp__', '__coerce__', '__delat
tr__', '__div__', '__divmod__', '__doc__', '__float__', '__floordiv__', '__forma
t__', '__getattribute__', '__getnewargs__', '__hash__', '__hex__', '__index__',
'__init__', '__int__', '__invert__', '__long__', '__lshift__', '__mod__', '__mul
__', '__neg__', '__new__', '__nonzero__', '__oct__', '__or__', '__pos__', '__pow
__', '__radd__', '__rand__', '__rdiv__', '__rdivmod__', '__reduce__', '__reduce_
ex__', '__repr__', '__rfloordiv__', '__rlshift__', '__rmod__', '__rmul__', '__ro
r__', '__rpow__', '__rrshift__', '__rshift__', '__rsub__', '__rtruediv__', '__rx
or__', '__setattr__', '__sizeof__', '__str__', '__sub__', '__subclasshook__', '_
_truediv__', '__trunc__', '__xor__', 'bit_length', 'conjugate', 'denominator', '
imag', 'numerator', 'real']
```

- Esempi di invocazione di metodi su valori numerici e su stringhe

```
>>> (-5).__abs__()
5
>>> ('Pippo').upper()
'PIPPO'
```

```
>>> (3+5j).real
3.0
>>> (3+5j).imag
5.0
```

Sul concetto di variabile in Python

- In linguaggi come C, C++, Java, che adottano un controllo del tipo dei dati statico (*a tempo di compilazione*) una *variabile* è un "contenitore di memoria" atto a mantenere un dato di un certo tipo
  - In C, la dichiarazione: `int a;`
    dice al compilatore di riservare un'area di memoria atta a contenere un dato di tipo intero (es. 32 bit), ed usa il nome a per identificare quest'area di memoria (es. un indirizzo di memoria centrale)
- In Python, il modello di programmazione è significativamente diverso: il linguaggio effettua un controllo dinamico dei tipi (*a tempo di esecuzione* o *runtime*)
- Un qualunque oggetto è associabile ad un "nome" (un identificatore) mediante l'operatore di assegnazione: `a = 5`
- Lo stesso "nome" può essere associato (*binding*) in momenti successivi ad oggetti diversi, anche di tipi diversi: `a = 5; a = "Pippo"`
  - Questi nomi sono tipicamente chiamati "variabili" in Python
- Un identificatore valido in Python è una sequenza di caratteri scelti tra lettere, cifre numeriche decimali ed il carattere underscore (_), con il vincolo che il primo carattere non può essere una cifra numerica
  - Identificatori validi in Python: temp, Cognome, p0, p123_, a_b, _3a
  - Identificatori NON validi in Python: 0p, papà, a-b, $3a
- Un nome non può essere usato prima che sia stato associato ad un valore

Tipizzazione dinamica

- In Python non si effettua la dichiarazione esplicita del tipo di una variabile
- L'interprete determina a runtime il tipo di una variabile in base al valore assegnatole
- Il tipo associato ad una variabile può cambiare nel corso dell'esecuzione
  - Si parla pertanto di *tipizzazione dinamica*
- La funzione predefinita `type()` restituisce il tipo di una variabile

```
>>> x = "Pippo"; print type(x)
<type 'str'>
>>> x = 123; print type(x)
<type 'int'>
>>> x = 3.14; print type(x)
<type 'float'>
>>> x = 3+5j; print type(x)
<type 'complex'>
```

Numeri

- Il linguaggio Python supporta nativamente quattro diversi tipi numerici:

  `int` `long` `float` `complex`

- Se nella rappresentazione è presente il punto decimale, il numero si intende *float*, altrimenti si intende di tipo *int* o *long*
  - Il tipo *int* è soggetto ad un limite di rappresentazione di macchina (tipicamente 32 bit)
  - Il tipo *long* può rappresentare numeri interi con rappresentazione illimitata
  - Un'espressione aritmetica con dati di tipo `int` può produrre un risultato di tipo `long` se il risultato non è rappresentabile come `int`
- Di default, i numeri interi sono rappresentati in notazione decimale
  - I numeri interi possono essere anche rappresentati dal programmatore in notazione:: ottale (la sequenza di cifre inizia con 0) o esadecimale (sequenza di cifre inizia con 0x)
- Il valore di un numero `float` può essere rappresentato in un programma sia in notazione con la virgola sia in notazione esponenziale
  - I numeri `float` sono rappresentati internamente secondo lo standard IEEE-754
- Esempi:
  - 12 numero intero in rappr. decimale
  - 012 numero intero in rappr. ottale (valore dieci)
  - 0x12 numero intero in rappr. esadecimale (valore diciotto)
  - 0.050143 numero floating point
  - 5.0143e-2 numero floating point in rappr. esponenziale
- Il tipo `complex` rappresenta numeri complessi; la parte reale ed il coefficiente immaginario sono numeri di tipo `real`
  - `a = 3 + 5j; print a.real, a.imag` produce l'output: `3.0 5.0`
- Il linguaggio è arricchito con librerie che consentono di trattare altri tipi numerici

Operatori aritmetici

- In Python esistono i classici operatori aritmetici per le operazioni aritmetiche fondamentali:

+	somma
-	sottrazione
*	prodotto
/	divisione
//	divisione intera
**	elevamento a potenza

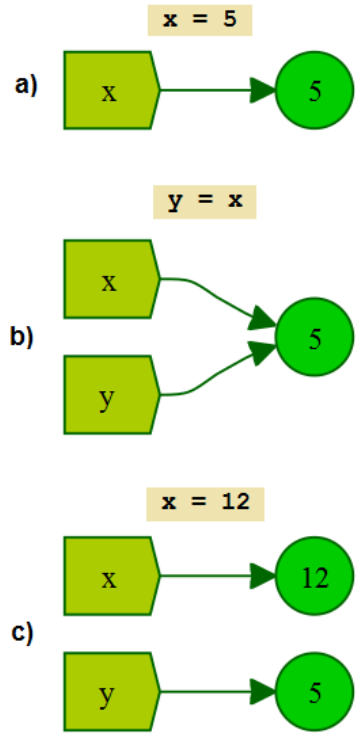
- Non esistono gli operatori aritmetici unari ++ e -- del C/C++
- Se gli operandi sono interi, il risultato di un'operazione aritmetica è un intero
- Se almeno uno degli operandi è float, il risultato è un float
- Valgono le usuali regole di priorità tra gli operatori
- Attraverso le parentesi si modifica l'ordine di applicazione degli operatori

```
>>> 3/2
1
>>> 3.0/2
1.5
>>> 3.0//2
1.0
>>> 3**2
9
>>> (3 + 2) * 7
35
```

Statement di assegnazione

- In generale, l'operatore di assegnazione crea un'associazione (*binding*) tra un nome ed un 'oggetto'
- La funzione predefinita `id(my_var)` restituisce un identificativo associato all'area di memoria che contiene il valore associato al nome `my_var`
- Si consideri quanto illustrato nell'esempio seguente

```
>>> x = 5
>>> y = x
>>> print id(x), id(y)
30696568 30696568
>>> x = 12; print x
12
>>> print y
5
>>> print id(x), id(y)
30696544 30696568
```



- L'esempio mostra che, nel momento in cui un nome (nell'esempio `x`) viene assegnato ad un nuovo valore (`x = 12`), cambia l'area di memoria associata al nome
  - Questo è molto diverso rispetto a quanto succede in C/C++

Ulteriori informazioni sull'operatore di assegnazione

- E' possibile realizzare *assegnazioni multiple* con un solo statement

  `a, b = 3, 5` equivale a: `a = 3; b = 5`

- E' possibile associare ad un nome il risultato di un'espressione
- Come in C, esistono gli operatori di assegnazione

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

  - `a += 1` equivale a: `a = a + 1`

```
>>> a, b, c = 1, 2, 3
>>> print a, b, c
1 2 3
>>> a += 2
>>> print a
3
>>> a **= 4
>>> print a
81
```

Funzioni

- Una funzione è un *sottoprogramma* al quale possono essere passati dei parametri (*argomenti*) e che eventualmente restituisce un valore
- Per definire una funzione si usa lo statement `def`
- Una funzione che non prevede argomenti ha una lista di argomenti vuota ()
- Per ritornare al chiamante un oggetto `x` si usa lo statement `return x`
- Un oggetto speciale `None` è ritornato da una funzione se:
  - il flusso di esecuzione della funzione termina senza aver eseguito un'istruzione *return*
  - lo statement `return` viene eseguito senza argomenti

```
def max(x, y):
    print "x = ", x
    print "y = ", y
    if (x >= y):
        return x
    else:
        return y

max(0, 0); print ""
ret = max(1, 1)
print "Valore di ritorno =", ret
ret = max(2, 1)
print "Valore di ritorno =", ret
s1 = "PIPPO"
s2 = "PIPPONE"
s3 = "PLUTO"
ret = max(s1, s2)
print "Valore di ritorno =", ret
ret = max(s2, s3)
print "Valore di ritorno =", ret
```

```
x =  0
y =  0

x =  1
y =  1
Valore di ritorno = 1
x =  2
y =  1
Valore di ritorno = 2
x =  PIPPO
y =  PIPPONE
Valore di ritorno = PIPPONE
x =  PIPPONE
y =  PLUTO
Valore di ritorno = PLUTO
```

Funzioni: argomenti, variabili locali e variabili globali

- Una funzione può avere delle variabili locali, il cui scope è limitato alla funzione
  - Una variabile locale può avere lo stesso nome di una variabile definita esternamente
- Gli argomenti di una funzione sono trattati alla stregua di variabili locali
  - Se nella funzione si modifica il valore associato ad un argomento, la modifica non si riflette su un'eventuale variabile esterna con lo stesso nome
- Per modificare in una funzione il valore associato ad un nome di variabile definito esternamente, si usa lo statement `global`

```
def f1(x):
    a = 3     # variabile locale
    x += 1    # argomento
    print "In f1:"
    print "a =", a, "x =", x

a = 99
x = 1
print "Prima di eseguire f1:"
print "a =", a, "x =", x
f1(x)
print "Dopo aver eseguito f1:"
print "a =", a, "x =", x
```

```
Prima di eseguire f1:
a = 99 x = 1
In f1:
a = 3 x = 2
Dopo aver eseguito f1:
a = 99 x= 1
```

```
def f2(x):
    global a
    a = 3     # variabile globale
    x += 1    # argomento
    print "In f2:"
    print "a =", a, "x =", x

a = 99
x = 1
print "Prima di eseguire f2:"
print "a =", a, "x =", x
f2(x)
print "Dopo aver eseguito f2:"
print "a =", a, "x =", x
```

```
Prima di eseguire f2:
a = 99 x = 1
In f2:
a = 3 x = 2
Dopo aver eseguito f2:
a = 3 x = 1
```

Funzioni: argomenti con valore di default

- Per ogni argomento di una funzione si può specificare un valore di default
- In questo caso, l'argomento diventa opzionale
- Gli argomenti per i quali non si specifica un valore di default sono obbligatori

```
def f(x, y = 5, z = 3):
    print "In f:",
    print "x =", x, "y =", y, "z =", z

f(0, 1, 2)
f(0)
f(8, 1)
f(7, z = 2)
f(6, y = 2)
f(0, z = 2, y = 1)
```

```
In f: x = 0 y = 1 z = 2
In f: x = 0 y = 5 z = 3
In f: x = 8 y = 1 z = 3
In f: x = 7 y = 5 z = 2
In f: x = 6 y = 2 z = 3
In f: x = 0 y = 1 z = 2
```

- Quando ad un argomento è assegnato un valore di default, nella chiamata della funzione, un parametro può essere specificato con la notazione `nome=valore`
  - In tal modo i parametri possono essere passati in ordine diverso da quello con cui sono elencati nella definizione della funzione

Funzioni di conversione

- E' possibile associare il valore di un dato ad un valore "corrispondente" di un altro tipo mediante apposite funzioni predefinite
- `int()` restituisce un `int` a partire da una stringa di caratteri o un `float`
- `float()` restituisce un `float` a partire da una stringa di caratteri o un `int`
- `int(x, b)` restituisce un `int` a partire dalla stringa `x` interpretata come sequenza di cifre in base `b`
- Di seguito si mostrano alcuni esempi

```
>>> int('2014')
2014
>>> int(3.141592)
3
>>> float('1.99')
1.99
>>> float(5)
5.0
>>> int('20',8)
16
>>> int('20',16)
32
>>> int('aa')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError:  invalid literal for int() with base 10:  'aa'
>>> int('aa',16)
170
```

La funzione round()

- La funzione round(number[, ndigits])
  restituisce un numero che arrotonda il valore di number con una precisione di ndigits cifre decimali
  - round restituisce sempre un float
  - Se non specificato, ndigits vale 0 per default

```
>>> round(1.22, 1)
1.2
>>> round(1.49)
1.0
>>> round(1.50)
2.0
```

Funzioni per l'input da tastiera

- La funzione `raw_input()` consente di leggere una stringa dallo standard input (tastiera)
- La funzione `input()` consente di leggere un numero (intero o float) in base 10 dallo standard input (tastiera)

```
x = input("Digita un numero e premi ENTER: ")
s = raw_input("Digita una stringa e premi ENTER: ")
```

- Se il valore fornito a `input()` non è un numero, si genera un errore a runtime

```
>>> x = input("Digita un numero e premi ENTER: ")
Digita un numero e premi ENTER: xxx
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<string>", line 1, in <module>
NameError:  name 'xxx' is not defined
>>>
```

Tipo bool

- In Python il tipo `bool` ha due valori: `True` e `False`
- E' possibile associare ad una variabile un valore `bool`
- IMPORTANTE: In Python si considerano `False` i seguenti valori:
  - `None`
  - `False`
  - zero, di un qualunque tipo numerico: 0, 0L, 0.0, 0j
  - strutture dati vuote: `''`, `()`, `[]`, `{}`

```
>>> print type(True); print type(False)
<type 'bool'>
<type 'bool'>
```

Operatori and, or e not

- Gli operatori booleani `and`, `or` e `not` sono definiti come segue

Operatore	*Valore restituito*
`a or b`	b, se a è `False` o un valore assimilato a `False` a, altrimenti
`a and b`	a, se a è `False` o un valore assimilato a `False` b, altrimenti
`not a`	`True`, se a è `False` o un valore assimilato a `False` `False`, altrimenti

- Ordine di priorità decrescente: `not`, `and`, `or`
  - `not a and b or c` equivale a: `((not a) and b) or c`
- `not` ha priorità minore di altri operatori non-booleani
  - `not a == b` equivale a `not (a==b)`

```
>>> True and False
False
>>> True or False
True
>>> not True
False
>>> not False
True
```

```
>>> 3 and 5
5
>>> 0 and 5
0
>>> True and 5
5
>>> False and 5
False
```

```
>>> 3 or 5
3
>>> 0 or 5
5
>>> True or 5
True
>>> False or 5
5
```

Espressioni di confronto

- Un'espressione di confronto restituisce un valore di tipo `bool`
- Usa gli operatori di confronto:

  `==` `<` `<=` `>=` `>` `!=` `is` `is not`

- I valori booleani `False` e `True` sono uguali agli interi 0 ed 1 rispettivamente
- Se necessario, nel calcolare un'espressione di confronto vengono eseguite conversioni di tipo `int` $\rightarrow$ `float`

```
>>> False == 0
True
>>> False == 1
False
>>> False == 2
False
>>> True == 0
False
>>> True == 1
True
>>> True == 2
False
>>> 1 == 2 - 1
True
>>> 1 == 1.0
True
>>> 1 == "1"
False
```

Statement if

- `if (cond)` esegue una sequenza di istruzioni se cond ha valore **diverso da** `False`
- Si consideri il seguente esempio:

```
>>> a=2
>>> if (a):
...     print "eseguo if"
... else:
...     print "eseguo else"
...
eseguo if
>>> if (a==True):
...     print "eseguo if"
... else:
...     print "eseguo else"
...
eseguo else
```

Statement if (2)

```
a = input("Digita un numero e premi ENTER: ")
print "Hai digitato",
if (a > 0):
    print "un numero POSITIVO (>0)"
elif (a < 0):
    print "un numero NEGATIVO (<0)"
else:
    print "ZERO"
```

- Le parentesi intorno alla condizione non sono necessarie
- La parte `elif` e la parte `else` sono facoltative
- Si possono aggiungere un numero di arbitrario di `elif`
- Lo statement `pass` rappresenta un'istruzione che non produce effetto
- Esempio d'uso di `pass`: lo statement C/C++

```
if (a) {} /* Se a fai niente */
b = 5;    /* Istruzione eseguita sempre */
```

si codifica in Python come segue:

```
if (a):
    pass
b = 5
```

Statement while

- while esegue un ciclo finché una condizione è vera

```
a = input("Digita un numero e premi ENTER (0 per terminare): ")
while (a != 0):
    print "Hai digitato ", a
    a = input("Digita un numero e premi ENTER (0 per terminare): ")

print "Hai digitato 0 ed il programma termina."
```

- Lo statement break nel corpo del ciclo fa uscire dal ciclo
- Lo statement continue nel corpo del ciclo salta all'iterazione successiva

```
conta = 0
while (True):
    a = input("Digita un numero e premi ENTER (0 per terminare): ")
    if (a > 0):
        print "Hai digitato un numero positivo: viene contato"
    elif (a < 0):
        print "Hai digitato un numero negativo: NON viene contato"
        continue
    else:
        print "Hai digitato 0 ed il programma termina."
        break
    conta = conta + 1

print "Hai digitato %d numeri positivi" % conta
```

Strutture dati *container* in Python

- Uno degli aspetti peculiari del linguaggio Python è la ricchezza di strutture dati definite dal linguaggio
- In particolare, il linguaggio offre al programmatore diverse strutture dati di tipo *container* atte a contenere oggetti di vario tipo
  - liste, tuple, dizionari, stringhe, set
  - il modulo `collections` definisce ulteriori tipi container
- Le varie strutture dati differiscono in vari aspetti:
  - il modo con il quale si può accedere agli oggetti contenuti
  - la possibilità di iterare sugli elementi contenuti nella struttura dati
  - l'eventuale ordinamento definito tra gli elementi contenuti
  - la possibilità o meno di modificare gli elementi presenti nella struttura dati una volta che sia stata "costruita"
    - si parla di strutture dati *mutabili* o *immutabili*
    - liste, dizionari, set sono mutabili mentre stringhe e tuple sono immutabili

In Python le stringhe sono immutabili. Dunque:

- `s = "pippo"; s[0] = 'P'` produce un errore: non è possibile modificare un carattere di una stringa in maniera diretta;
- `s = "Pippo"; s = s + " e Pluto"` produce l'allocazione a runtime di tre stringhe: `"Pippo"`, `" e Pluto"`, e `"Pippo e Pluto"`

Tipo stringa

- In Python è possibile definire una stringa di caratteri usando come delimitatori:
  apici `'esempio'` doppi apici `"esempio"`
- Tre apici `'''` e tre doppi apici `"""` sono usati per stringhe su più linee
- Caratteri speciali possono essere inseriti in una stringa mediante sequenze di escape
- La concatenazione di stringhe si effettua con l'operatore +
- L'operatore * consente la concatenazione di una stringa a se stessa per *n* volte
- E' possibile identificare mediante un indice intero i singoli caratteri di una stringa s
  - s[0] indica il primo, s[1] il secondo, ecc. ...
- E' possibile anche usare valori negativi per l'indice:
  - s[-1] indica l'ultimo carattere, s[-2] il penultimo, ecc. ...

```
>>> print 'riga 1\nriga 2'
riga 1
riga 2
>>> nome = "Roberto"
>>> print "Ciao "+nome
Ciao Roberto
>>> print "Ciao"*3
CiaoCiaoCiao
>>> s = '''Questa è una stringa
... che contiene
...     tre linee'''
>>> s1 = "ABC"; s1[0] = 'X'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  'str' object does not support item assignment
```

```
>>> s = "ABCDEFGHIJ"
>>> print s[0]; print s[1]
A
B
>>> print s[9]
J
>>> print s[10]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError:  string index out of range
>>> print s[-1]; print s[-2]
J
I
>>> print s[-10]
A
```

Inserimento di valori variabili in stringhe costanti

- E' possibile combinare una stringa di testo fatta da una parte costante specificata tra apici ed una parte definita attraverso i nomi di variabili
- Come si può fare in C per la funzione *printf*, ciò si realizza inserendo nella stringa costante il simbolo % come placeholder, seguito da dei codici che servono a definire la formattazione dell'output prodotto
- I valori da sostituire ai placeholder sono indicati a destra della stringa dopo %
- Esempio con un solo valore:

```
>>> import math
>>> print "The value of PI is approximately %5.3f." % math.pi
The value of PI is approximately 3.142.
```

- Esempio con due valori:

```
>>> x=18; y=15
>>> print "x=%d y=%d" % (x,y)
x=18 y=15
```

Stringhe: slicing

E' possibile costruire una nuova stringa prendendo dei "pezzi" di una stringa formati da caratteri consecutivi

- Si usa la notazione con indici, specificando l'indice `i` del primo carattere (compreso) e quello `j` dell'ultimo (escluso) separati dal carattere `:` (due punti)

  `s[i:j]`

- Se l'estremo sinistro `i` è omesso: `s[:j]`

  la sottostringa inizia dal primo carattere di `s`

- Se l'estremo destro `j` è omesso: `s[i:]`

  la sottostringa termina con l'ultimo carattere di `s`

```
>>> s = "Qui, Quo, Qua"
>>> s[0:3]
'Qui'
>>> s[:3]
'Qui'
>>> s[3:3]
''
>>> s[3:4]
','
>>> s[5:8]
'Quo'
>>> s[-3:-1]
'Qu'
>>> s[-3:]
'Qua'
>>>
```

Stringhe: metodi (1)

Una stringa è un *oggetto* sul quale si possono invocare dei *metodi* predefiniti

- `s.find(sub_str)` restituisce l'indice della posizione della prima occorrenza della sottostringa `sub_str` nella stringa `s`
- `s.find(sub_str, start)` restituisce l'indice della posizione della prima occorrenza della sottostringa `sub_str` nella stringa `s`, cominciando la ricerca dal carattere di indice `start`
  - `find()` restituisce -1 se la sottostringa `sub_str` non è trovata
- `s.split(sep)` restituisce una lista di sottostringhe di `s` separate nella stringa originaria dal separatore `sep`
  - Se `sep` non esiste in `s`, `split()` restituisce una lista con il solo elemento `s`

```
>>> s = "Qui, Quo, Qua"
>>> s.find(", ")
3
>>> s.find(", ",0)
3
>>> s.find(", ",3)
3
>>> s.find(", ",4)
8
>>> s.find("X")
-1
>>> s.split(",")
['Qui', ' Quo', ' Qua']
>>> s.split(", ")
['Qui', 'Quo', 'Qua']
>>> s.split("X")
['Qui, Quo, Qua']
```

Stringhe: metodi (2)

- `s.strip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità sinistra e destra
- `s.rstrip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità destra
- `s.lstrip()` restituisce una stringa ottenuta eliminando da `s` i caratteri spazio, tab (`\t`), newline (`\n`) posti all'estremità sinistra
- `s.startswith(x)` restituisce `True` se la stringa `s` inizia con la sottostringa `x`, `False` altrimenti
- `s.endswith(x)` restituisce `True` se la stringa `s` termina con la sottostringa `x`, `False` altrimenti
- `s.upper(x)` restituisce una stringa in cui i caratteri di `s` che sono lettere minuscole sono convertiti in maiuscole, gli altri caratteri sono lasciati inalterati
- `s.lower(x)` restituisce una stringa in cui i caratteri di `s` che sono lettere maiuscole sono convertiti in minuscole, gli altri caratteri sono lasciati inalterati

```
>>> s = "  Qui, \tQuo, \tQua\t\n"
>>> print s
  Qui,  Quo,    Qua

>>> s.strip()
'Qui, \tQuo, \tQua'
>>> s.rstrip()
'  Qui, \tQuo, \tQua'
>>> s.lstrip()
'Qui, \tQuo, \tQua\t\n'
```

```
>>> s = "Qui, Quo, Qua"
>>> s.startswith('Qui')
True
>>> s.endswith('Qua')
True
>>> s.upper()
'QUI, QUO, QUA'
>>> s.lower()
'qui, quo, qua'
>>>
```

Liste

- Le liste sono strutture dati ordinate che possono contenere oggetti di tipi differenti
- Il seguente statement crea una lista associata al nome `l`:
  `l = [1, 2, "Pippo"]`
- La scrittura `[]` indica una *lista vuota*
- Sulle liste si possono applicare gli operatori + e * come per le stringhe
- La funzione `len(l)` restituisce il numero di elementi di una lista `l`
- Gli elementi di una lista possono essere individuati e modificati tramite un indice come per gli array in C/C++
- La scrittura `l[i:j]` indica la lista costituita dagli elementi di `l` compresi tra quello di indice `i` (compreso) a quello di indice `j` (escluso)
  - Se `i` non è specificato, si intende `i=0`
  - Se `j` non è specificato, si intende `j=len(l)`

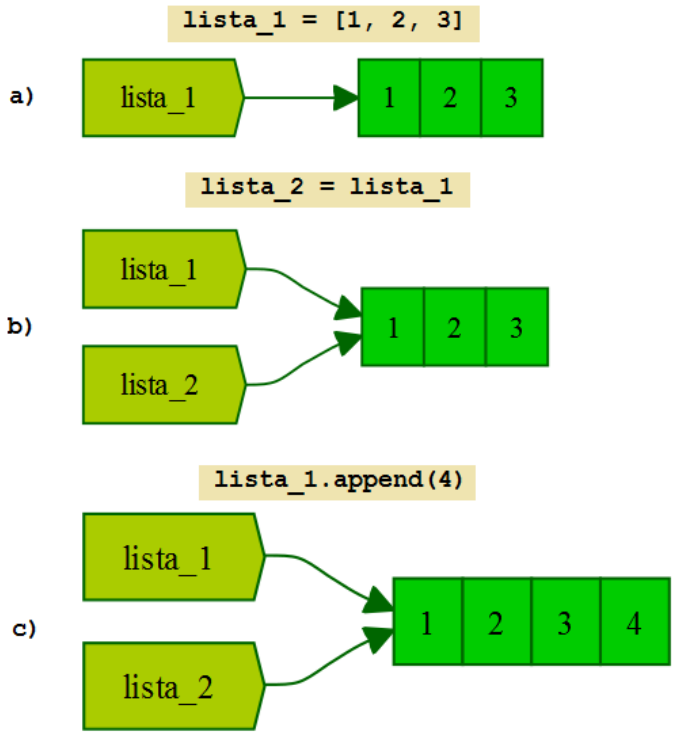
```
>>> l = [1, 2]
>>> m = ["Pippo", 3, 4]
>>> print l + m
[1, 2, 'Pippo', 3, 4]
>>> print l*4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> print m[0]
Pippo
>>> m[0] = 99; print m
[99, 3, 4]
>>> len(m)
3
```

```
>>> n = l + m[1:]; print n
[1, 2, 3, 4]
>>> print n[1:2]
[2]
>>> print n[1:3]
[2, 3]
>>> print n[1:4]
[2, 3, 4]
>>> print n[1:5]
[2, 3, 4]
>>> print n[5:6]
[]
```

Le liste sono oggetti mutabili

- Si consideri quanto illustrato nell'esempio seguente

```
>>> lista_1 = [1, 2, 3]
>>> lista_2 = lista_1
>>> print id(lista_1), id(lista_2)
32782280 32782280
>>> lista_1.append(4); print lista_1
[1, 2, 3, 4]
>>> print lista_2
[1, 2, 3, 4]
>>> print id(lista_1), id(lista_2)
32782280 32782280
```



- L'esempio mostra che l'invocazione del metodo `append()` su un oggetto di tipo lista (nell'esempio `lista_1`) non cambia l'area di memoria associata al nome
  - Di conseguenza, la modifica operata su `lista_1` si riflette anche su `lista_2`

Liste: metodi

- Una lista è un *oggetto* sul quale si possono invocare dei *metodi* predefiniti:
  - `l.append(obj)` aggiunge `obj` "in coda" alla lista `l`
  - `l.extend(l1)` estende `l` aggiungendo "in coda" gli elementi della lista `l1`
  - `l.insert(index,obj)` aggiunge `obj` in `l` prima della posizione indicata da `index`
  - `l.pop(index)` rimuove da `l` l'oggetto nella posizione `index` e lo restituisce
  - `l.remove(obj)` rimuove la prima occorrenza di `obj` nella lista `l`
  - `l.reverse()` dispone gli elementi della lista `l` in ordine inverso
  - `l.sort()` dispone gli elementi della lista `l` in ordine crescente
  - `l.index(obj)` restituisce l'indice della prima occorrenza di `obj` nella lista `l`
  - `l.count(obj)` restituisce il numero di occorrenze di `obj` nella lista `l`

```
>>> l1 = [6, 7, 8, 9]
>>> l2 = [1, 2, 3, 4, 5]
>>> l1.append(10); print l1
[6, 7, 8, 9, 10]
>>> l1.extend(l2); print l1
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(0, "START"); print l1
['START', 6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
>>> l1.insert(6, "x"); print l1
['START', 6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.pop(0); print l1
'START'
[6, 7, 8, 9, 10, 'x', 1, 2, 3, 4, 5]
>>> l1.remove("x"); print l1
[6, 7, 8, 9, 10, 1, 2, 3, 4, 5]
```

```
...
>>> l1.reverse(); print l1
[5, 4, 3, 2, 1, 10, 9, 8, 7, 6]
>>> l1.sort(); print l1
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> l1.pop(-1)
10
>>> l1.index(1)
0
>>> l1.index(9)
8
>>> l1.index(10)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError:  10 is not in list
```

Altre funzioni predefinite: `range()`, `len()`, `max()`

- La funzione `range(n)`
  restituisce una lista formata dai numeri interi compresi tra 0 ed $n-1$
- La funzione `range(m, n, step)`
  restituisce una lista formata dai numeri interi compresi tra
  m (incluso) ed n (escluso), con un incremento di *step*
- La funzione `len(x)`
  restituisce il numero di elementi di una lista `x` (si può applicare anche a tuple e stringhe)
- La funzione `max(l)`
  restituisce l'elemento massimo di una lista `l`

```
>>> range(5)
[0, 1, 2, 3, 4]
>>> range(1,6,2)
[1, 3, 5]
>>> len([0, 1, 2, 3])
4
>>> max([1.0, 2.5, -2.3, 1.3])
2.5
```

Unpacking di liste mediante assegnazioni multiple

- Un uso particolare dello statement di assegnazione multipla: scompattare una lista
  - `a, b = [1, 2]` produce `a = 1; b = 2`
  - Nota: il numero di variabili a sinistra dell'operatore di assegnazione deve essere uguale alla dimensione della lista
- Si può usare l'assegnazione multipla per scambiare due variabili
  - `a, b = b, a` scambia i nomi delle variabili `a` e `b`

```
>>> v = [1, 2]
>>> a, b = v
>>> print a
1
>>> print b
2
>>> a, b = b, a
>>> print a
2
>>> print b
1
>>> v = [1, 2, 3, 4]
>>> a, b = v
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError:  too many values to unpack
```

Funzioni: argomenti con valore di default mutabile (1)

- Il valore di default di un argomento di una funzione viene valutato solo una volta, nel momento in cui l'interprete trova lo statement di definizione della funzione
- Se il valore di default di un argomento è un **oggetto mutabile**, l'oggetto viene creato una sola volta e una modifica del valore dell'argomento si riflette su tutte le successive chiamate della funzione

```
conta_chiamate = 0

def f(x = 1, y = []):
    global conta_chiamate
    conta_chiamate += 1
    print "In f - volta n.", conta_chiamate, ":"
    print "x =", x, "y =", y
    x = 0
    y.append(conta_chiamate)
    print "In f - modifica x ed y:"
    print "x =", x, "y =",y,"\n------------------"

for i in range(0,3):
    f()
```

```
In f - volta n. 1 :
x = 1 y = []
In f - modifica x ed y:
x = 0 y = [1]
------------------
In f - volta n. 2 :
x = 1 y = [1]
In f - modifica x ed y:
x = 0 y = [1, 2]
------------------
In f - volta n. 3 :
x = 1 y = [1, 2]
In f - modifica x ed y:
x = 0 y = [1, 2, 3]
------------------
```

Funzioni: argomenti con valore di default mutabili (2)

- Se il comportamento di default desiderato è quello di inizializzare l'argomento non specificato ad una lista vuota, occorre assegnare come valore di default `None` e poi riconoscere questa situazione nel codice

```
conta_chiamate = 0

def f(x = 1, y = None):
    global conta_chiamate
    conta_chiamate += 1
    print "In f - volta n.", conta_chiamate, ":"
    print "x =", x, "y =", y
    x = 0
    if (y is None): y = []
    y.append(conta_chiamate)
    print "In f - modifica x ed y:"
    print "x =", x, "y =",y,"\n------------------"

for i in range(0,3):
    f()
```

```
In f - volta n. 1 :
x = 1 y = None
In f - modifica x ed y:
x = 0 y = [1]
------------------
In f - volta n. 2 :
x = 1 y = None
In f - modifica x ed y:
x = 0 y = [2]
------------------
In f - volta n. 3 :
x = 1 y = None
In f - modifica x ed y:
x = 0 y = [3]
------------------
```

Tuple

- Le tuple sono strutture dati ordinate che possono contenere oggetti di tipi differenti
- Il seguente statement crea una tupla associata al nome `t`:
  `t = (1, 2, "Pippo")`
- `()` indica una *tupla vuota*, `(a,)` indica una tupla con il solo elemento `a`
- Sulle tuple si possono applicare gli operatori `+` e `*` come per le stringhe
- La funzione `len(t)` restituisce il numero di elementi di una tupla `t`
- Gli elementi di una tupla possono essere individuati (**ma non modificati**) tramite un indice come per gli array in C/C++
- La scrittura `t[i:j]` indica la tupla costituita dagli elementi di `t` compresi tra quello di indice `i` (compreso) a quello di indice `j` (escluso)
  - Se `i` non è specificato, si intende `i=0`
  - Se `j` non è specificato, si intende `i=len(t)`

```
>>> t1 = (1, 2)
>>> t2 = ("Pippo", 3, 4)
>>> t = t1 + t2; print t
(1, 2, 'Pippo', 3, 4)
>>> len(t)
5
>>> print t[0]
1
>>> print t[2]
Pippo
```

```
>>> t[0] = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError:  'tuple' object does not support item assignment
>>> print t[1:3]
(2, 'Pippo')
>>> print t[1:1]
()
>>> print t[5:5]
()
```

Confronto tra strutture contenitore

- E' possibile usare gli operatori di confronto per confrontare dati di tipo *container*
- Il confronto è di tipo lessicografico, con i seguenti criteri
- Strutture dello stesso tipo sono confrontate elemento per elem., a partire dal primo
  - se gli elementi sono tutti rispettivamente uguali, le strutture sono considerate uguali
    - `[1, 2] == [1, 2]` è `True`
    - `[1, 2] == [True, 2.0]` è `True`
  - il primo elemento diverso determina quale delle due strutture è maggiore dell'altra
    - `[1, 2, 3] > [1, 2, 2]` è `True`
    - `[1, 2, 3] > [1, 2]` è `True`
    - `[1, 1, 3] < [1, 2]` è `True`
    - `(1, 2) < (1, 2, -1)` è `True`
    - `"Pippo" < "Pippozzo"` è `True`
- Due strutture di tipo diverso sono confrontate per nome di tipo:

```
list < string < tuple
```

  - `(1, 2) > [3, 4]` è `True`
  - `[3, 4] < (1, 2)` è `True`
  - `[1, 2, 3] < "Pippo"` è `True`
  - `"Pippo" < (1, 2)` è `True`

Statement `for`

- Lo statement `for` esegue una sequenza di istruzioni (corpo del ciclo) per tutti gli elementi di una struttura dati *iterabile*
- La struttura generale è:

```
for iterating_var in sequence:
   corpo_del_ciclo
```

- Esempio

```
nomi = ['Antonio', 'Mario',  'Giuseppe', 'Francesco']
for nome in nomi:
   print 'Nome :', nome
```

- E' possibile iterare su una lista mediante un indice intero come nell'esempio seguente

```
nomi = ['Antonio', 'Mario',  'Giuseppe', 'Francesco']
for index in range(len(nomi)):
   print 'Nome :', nomi[index]
```

- `break` nel corpo del ciclo fa uscire dal ciclo
- `continue` nel corpo del ciclo salta all'iterazione successiva

Creazione di liste mediante `for`

- Lo statement `for` può essere usato per creare liste i cui elementi sono generati da un'espressione valutata iterativamente
- Esempi

```
>>> l1 = [ (2*x + 1) for x in range(0,11) ]
>>> print l1
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
>>> l2 = [(x, x**2) for x in range(0,11)]
>>> print l2
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25), (6, 36), (7, 49), (8, 64), (9, 81), (10, 100)]
>>> l3 = [chr(i) for i in range(ord('a'),ord('z')+1)]
>>> print l3
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's',
 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Dizionari

- Un dizionario è una struttura dati contenitore di coppie (chiave, valore) in cui ciascun valore è identificato univocamente da una "chiave"
- Il seguente statement crea un dizionario associato al nome `d`

```
d = {chiave1: val1, chiave2: val2, chiave3: val3}
```

- L'accesso agli elementi di un dizionario avviene fornendo il valore della chiave:

`d[key]`

- Se il valore di chiave non esiste nel dizionario, si produce un errore
- I dizionari sono strutture dati mutabili, il valore della chiave è immutabile
- L'operatore `in` restituisce `True` se una chiave è presente in un dizionario

```
>>> d = {"NA": "Napoli",    "AV": "Avellino",
         "BN": "Benevento", "CE": "Caserta",
         "SA": "Salerno"}
>>> d["BN"]
'Benevento'
>>> d["MI"]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError:  'MI'
>>> d[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError:  0
```

```
>>> d["NA"] = "Naples"
>>> print d["NA"]
Naples
>>> "NA" in d
True
>>> "AV" in d
True
>>> "MI" in d
False
```

Dizionari: metodi

- Un dizionario è un *oggetto* sul quale si possono invocare dei *metodi* predefiniti:
  - `d.clear()` elimina tutti gli elementi del dizionario `d`
  - `d.copy()` restituisce una copia del dizionario `d`
  - `d.has_key(key)` restituisce `True` se in `d` esiste la chiave `key`
  - `d.items()` restituisce una lista con le tuple `(chiave, valore)` in `d`
  - `d.keys()` restituisce una lista con le chiavi in `d`
  - `d.values()` restituisce una lista con i valori in `d`
  - `d.update(d2)` aggiunge al contenuto di `d` quello di `d2`
  - `d.get(key, val)` restituisce il valore associato a `key`, altrimenti `val`
  - `d.get(key)` restituisce il valore associato a `key`, altrimenti `None` `l`

```
>>> d={"NA":"Napoli","AV":"Avellino","BN":"Benevento","CE":"Caserta","SA":"Salerno"}
>>> d.has_key("NA")
True
>>> d.has_key("MI")
False
>>> d.items()
[('NA', 'Napoli'), ('BN', 'Benevento'), ('SA', 'Salerno'), ('CE', 'Caserta'), ('AV', 'Avellino')]
>>> d.keys()
['NA', 'BN', 'SA', 'CE', 'AV']
>>> d.values()
['Napoli', 'Benevento', 'Salerno', 'Caserta', 'Avellino']
>>> print d.get("NA")
Napoli
>>> print d.get("MI")
None
>>> d.update({"NA": "Naples", "MI": "Milano"})
>>> print d
{'AV': 'Avellino', 'NA': 'Naples', 'BN': 'Benevento', 'MI': 'Milano', 'SA': 'Salerno', 'CE': 'Caserta'}
```

La funzione predefinita `sorted()`

- La funzione predefinita `sorted()` accett in ingresso una qualunque struttura dati iterabile e restituisce una lista di valori ordinata
- E' opportuno non confondere la funzione `sorted()` con il metodo `sort()` invocabile su una lista
- Se si passa come argomento un dizionario, `sorted()` restituisce la lista ordinata delle chiavi
- Tramite l'argomento opzionale `reverse` è possibile ordinare in ordine decrescente
- Esempi

```
>>> l = [3, 2, 5, 4, 7, 1]
>>> sorted(l)
[1, 2, 3, 4, 5, 7]
>>> print l
[3, 2, 5, 4, 7, 1]
>>> l.sort()
>>> print l
[1, 2, 3, 4, 5, 7]
>>> sorted(l, reverse=True)
[7, 5, 4, 3, 2, 1]
>>> t = ("Pippo", "Pluto", "Paperino")
>>> sorted(t)
['Paperino', 'Pippo', 'Pluto']
>>> d = {2:"Pippo", 3:"Pluto", 1:"Paperino"}
>>> sorted(d)
[1, 2, 3]
>>>
```

Gestione delle eccezioni

- Un'eccezione è un errore che si produce a tempo di esecuzione (*runtime*)
- Quando si verifica un errore, di regola il programma è terminato
- In alcune circostanze è possibile prevedere il verificarsi di un errore
  - ad es. perchè l'input fornito dall'utente non è corretto
- Con `try` è possibile "catturare" un evento di errore prodotto da uno statement
- Esempio (due versioni)

```
while True:
  try:
    x = int(raw_input("Inserisci un
          numero intero: "))
    break
  except ValueError:
    print "Non hai inserito un
          numero intero valido.
          Riprova."

print "Hai inserito il numero", x
```

```
while True:
  try:
    x = int(raw_input("Inserisci un
          numero intero: "))
  except ValueError:
    print "Non hai inserito un
          numero intero valido.
          Riprova."
    continue
  break

print "Hai inserito il numero", x
```

- Esempio di esecuzione

```
Inserisci un numero: x
Non hai inserito un numero intero valido. Riprova.
Inserisci un numero: 3.3
Non hai inserito un numero intero valido. Riprova.
Inserisci un numero: 33
Hai inserito il numero 33
```

Uso di librerie in Python: `import`

- Lo statement `import my_lib` dice all'interprete di rendere visibile (nello script in cui si trova) tutto ciò che è visibile a livello globale nel file `my_lib.py`
  - In questo modo si possono usare variabili e funzioni definite in una libreria
  - I nomi di variabili e funzioni della libreria sono associati al *namespace* `my_lib`
  - Il nome `n` definito in `my_lib.py` dovrà essere riferito come `my_lib.n` in uno script che fa `import my_lib`
  - Se lo script `my_lib.py` contiene statement eseguibili, essi sono eseguiti nel momento in cui è eseguito l'`import`
  - Il file libreria *my_lib.py* si può trovare:
    - o nella stessa cartella dove si trova lo script che fa l'`import`
    - o in una cartella prevista dall'interprete (es. in `C:\Python27\Lib\site-packages`)
- Lo statement `from my_lib import *` dice all'interprete di rendere visibili nel *namespace* dello script corrente tutti i nomi definiti nella libreria `my_lib.py`
  - In questo caso, il nome `n` definito in `my_lib.py` potrà essere direttamente riferito come `n` nello script che fa l'`import`
  - Occorre fare attenzione a possibili conflitti di nomi definiti in file differenti
- L'interprete Python rende già disponibili al programmatore molte librerie che implementano funzioni di utilità (si parla di *standard library* del linguaggio)
  - Ad es. `time`, `datetime`, `string`, `os`, `sys`, `getopt`, `fileinput`, ...
- Molte altre librerie Python possono essere scaricate ed installate con il tool `pip`
  - Ad es. `numpy`, `matplotib`, ...

Uso di import: esempio

```
# File: my_lib.py
currentYear = 2017

def currentAge(birthYear):
    global currentYear
    return (currentYear-birthYear)

print "Libreria my_lib caricata"
```

- L'esecuzione di `my_lib_test.py` produce come output:

```
Libreria my_lib caricata
Eta' dei miei amici nell'anno 2017
      Giovanni2 : eta' 41
          Mario : eta' 45
         Andrea : eta' 39
          Luigi : eta' 49
       Giovanni : eta' 52
```

```
# File: my_lib_test.py
import my_lib

# NB: le chiavi devono essere uniche
annoNascita_amici = { 'Mario': 1972,
                      'Giovanni': 1965,
                     'Giovanni2': 1976,
                        'Andrea': 1978,
                         'Luigi': 1968 }

print "Eta' dei miei amici nell'anno",
      my_lib.currentYear

for amico in annoNascita_amici:
    print "%15s : eta'" % amico,
    print my_lib.currentAge(
          annoNascita_amici[amico])
```

- Si osservi che gli elementi di un dizionario non sono memorizzati in un ordine particolare, pertanto l'ordine con il quale l'iteratore `in` restituisce gli elementi di `annoNascita_amici` non coincide con quello con il quale gli elementi sono stati scritti nello statement di assegnazione

Uso di `from-import`: esempio

```
# File: my_lib_test_2.py
from my_lib import *

# NB: le chiavi devono essere uniche
annoNascita_amici = { 'Mario': 1972,
                      'Giovanni': 1965,
                     'Giovanni2': 1976,
                        'Andrea': 1978,
                         'Luigi': 1968 }

print "Eta' dei miei amici nell'anno", currentYear

for amico in annoNascita_amici:
    print "%15s : eta'" % amico,
    print currentAge(annoNascita_amici[amico])
```

- L'output prodotto da questo programma è identico a quello prodotto dal precedente
- Si noti che i nomi `currentYear` e `currentAge` sono adesso nel namespace principale
  - La notazione `my_lib.currentYear` e `my_lib.currentAge` produrrebbe un errore

Esempi di programmi Python

Esempio #1: calcolo numeri primi

- Come esempio di funzione, si riporta sotto il codice di un programma che calcola i numeri primi compresi tra 1 e 1000 mediante una funzione

```
def primes(up_to):
    primes = []
    for n in range(2, up_to + 1):
        is_prime = True
        for divisor in range(2, n):
            if n % divisor == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(n)
    return primes

print primes(1000)
```

- L'output prodotto è mostrato sotto

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73,
 79, 83, 89, 97, 101, 103, 107, 109, 113, 127, 131, 137, 139, 149, 151, 157, 163
, 167, 173, 179, 181, 191, 193, 197, 199, 211, 223, 227, 229, 233, 239, 241, 251
, 257, 263, 269, 271, 277, 281, 283, 293, 307, 311, 313, 317, 331, 337, 347, 349
, 353, 359, 367, 373, 379, 383, 389, 397, 401, 409, 419, 421, 431, 433, 439, 443
, 449, 457, 461, 463, 467, 479, 487, 491, 499, 503, 509, 521, 523, 541, 547, 557
, 563, 569, 571, 577, 587, 593, 599, 601, 607, 613, 617, 619, 631, 641, 643, 647
, 653, 659, 661, 673, 677, 683, 691, 701, 709, 719, 727, 733, 739, 743, 751, 757
, 761, 769, 773, 787, 797, 809, 811, 821, 823, 827, 829, 839, 853, 857, 859, 863
, 877, 881, 883, 887, 907, 911, 919, 929, 937, 941, 947, 953, 967, 971, 977, 983
, 991, 997]
```

Esempio #1-bis: calcolo numeri primi

- Esercizio: determinare il numero di numeri primi presenti in ciascuna centinaia
  - I numeri primi compresi tra 0 e 99 sono 25
  - I numeri primi compresi tra 100 e 199 sono 21
  - ....
- Si estenda il codice dell'esempio 1
- Soluzione:

```
def primes(up_to):
    primes = []
    for n in range(2, up_to + 1):
        is_prime = True
        for divisor in range(2, n):
            if n % divisor == 0:
                is_prime = False
                break
        if is_prime:
            primes.append(n)
    return primes

p = primes(1000)
n = [0]*10
for i in range(len(p)):
   c = p[i] / 100
   n[c] += 1

print n
```

- L'output prodotto è:

```
[25, 21, 16, 16, 17, 14, 16, 14, 15, 14]
```

Esempio #2: funzione per la manipolazione di una stringa

- Siccome le stringhe sono oggetti *immutable*, quando occorre eseguire una manipolazione di una stringa, il programmatore ne deve creare una nuova
- Un tentativo di alterazione diretta dei caratteri di una stringa produce un errore
  - `s = "pippo"; s[0] = 'P'` produce l'errore:

    `'str' object does not support item assignment`
- Il codice riportato sotto mostra una funzione che restituisce una stringa a partire da una stringa fornita come primo argomento, nella quale si opera la sostituzione del carattere fornito come secondo argomento con il carattere fornito come terzo argomento (che per default è '~')

```
def replace(origin_string, char_to_replace, new_char='~'):
    new_string = ''
    for i in range(len(origin_string)):
        if (origin_string[i] == char_to_replace):
            new_string += new_char
        else:
            new_string += origin_string[i]
    return new_string

a = "Qui, Quo e Qua sono nipoti di Paperino"
print a
# produce come output: Qui, Quo e Qua sono nipoti di Paperino
print replace(a, ' ', '_')
# produce come output: Qui,_Quo_e_Qua_sono_nipoti_di_Paperino
print replace(a, ' ')
# produce come output: Qui,~Quo~e~Qua~sono~nipoti~di~Paperino
```

Esempio #3: analisi di dati estratti da file CSV (1)

- Si vuole scrivere un programma Python che estragga da un file dati testuale in formato CSV i voti in trentesimi conseguiti da un insieme di studenti e successivamente rappresenti con un istogramma la distribuzione dei voti nel campione
- La struttura del file *voti.csv* è la seguente:

```
Cognome,Nome,Voto,Lode
Amato,Alfredo,21,NO
Andreolli,Antonio,24,NO
Baresi,Carlo,19,NO
Carbonara,Francesco,27,NO
....
```

- Ogni riga contiene una sequenza di dati separati da virgole
- Il significato dei dati è descritto nella prima riga del file

Esempio #3: analisi di dati estratti da file CSV (2)

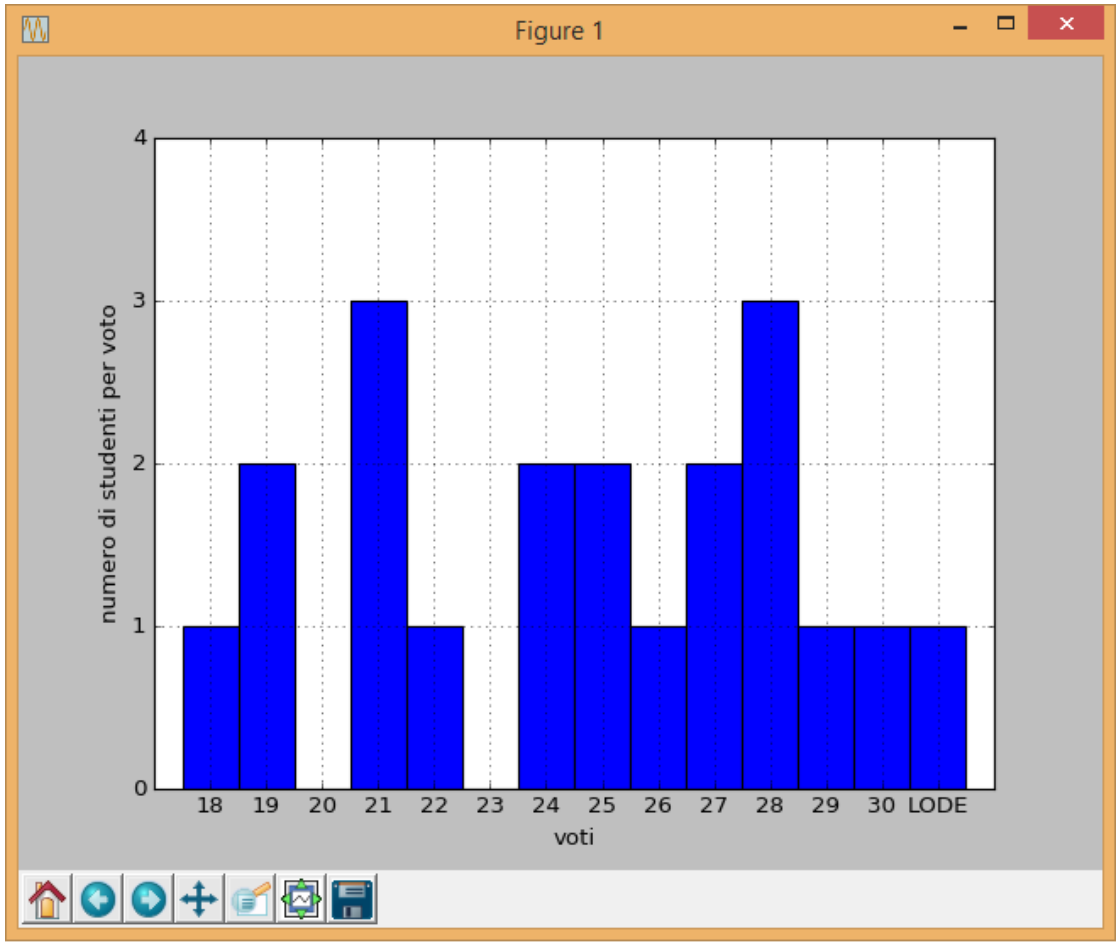
```
# File:  elabora-voti.py

csv_file = open("voti.csv", "r")
lines = csv_file.readlines()[1:]  # Ignora la prima linea
votes = []
for line in lines:
    line = line.rstrip('\n')
    data_item = line.split(",")
    cognome = data_item[0]
    nome = data_item[1]
    voto = int(data_item[2])
    lode = data_item[3]
    if (voto == 30) and (lode == "SI"):
        voto = 31    # 30 e lode si rappresenta come 31
    votes.append(voto)

csv_file.close()
....
```

Esempio #3: analisi di dati estratti da file CSV (3)

```
# File:  elabora-voti.py
....

import matplotlib.pyplot as plt
import numpy as np

hist, bin_edges = np.histogram(votes, bins=np.arange(18,33))

x = range(18,32)
plt.bar(x, hist, align='center', width=1)

x_labels = x[0:-1] + ['LODE']
plt.xticks(x, x_labels)

y_labels = range(0,max(hist)+1+1)
plt.yticks(y_labels)

plt.xlim(17, 32)
plt.ylim(0, max(hist)+1)
plt.xlabel('voti')
plt.ylabel('numero di studenti per voto')
plt.grid(True)

plt.show()
```

Esempio #3: analisi di dati estratti da file CSV (4)

- Il programma produce come output la figura seguente:



Programmazione ad oggetti in Python

Il concetto di classe in Python

- Una classe è un modello di oggetti costituito da attributi e metodi
- Relativamente agli attributi, occorre distinguere tra
  - *attributi di classe*, condivisi da tutte le istanze della classe
  - *attributi di istanza*, specifici per ciascuna istanza della classe
- Una classe `MyClass` è definita mediante uno statement `class`

```
class MyClass:
    statement_1
    statement_2
    statement_3
    ...
```

- Lo statement `class` crea un nuovo namespace `MyClass`
- Gli statement che costituiscono il corpo di `class` sono eseguiti
- Tipicamente, il corpo di `class` è costituito da statement del tipo:
  - `nome = valore` - assegnazioni per la inizializzazione di *attributi di classe*
  - `def nome_metodo:` - definizione di funzioni membro (*metodi*):
- Gli attributi di classe sono condivisi tra tutte le istanze della classe e sono identificati con la scrittura `MyClass.nome`

Costruttore e altri metodi

- Nella definizione di una classe, il metodo `__init__` ha la funzione di *costruttore*
- Esso viene eseguito quando si crea un'istanza di una classe mediante un'istruzione:

```
nome_oggetto = MyClass(param1, param2, ...)
```

- Nella definizione dei metodi della classe, incluso `__init__`, il primo argomento deve essere `self`, un riferimento all'istanza che è poi passato implicitamente all'atto dell'invocazione del metodo

```
def __init__(self, param1, param2, ...):
```

- Nel codice che costituisce il corpo dei metodi, incluso il costruttore, per fare riferimento agli attributi (*variabili di istanza*) si usa la notazione

```
self.nome_attributo
```

- La scrittura `nome_oggetto.f()` equivale a `MyClass.f(nome_oggetto)`
- La funzione predefinita `isinstance(obj, class)` restituisce `True` se `obj` è istanza della classe `class`, `False` altrimenti

```
>>> class MyClass1:
...     pass
...
>>> class MyClass2:
...     pass
...
>>> obj1 = MyClass1()
>>> obj2 = MyClass2()
```

```
>>> isinstance(obj1, MyClass1)
True
>>> isinstance(obj2, MyClass1)
False
>>> isinstance(obj1, MyClass2)
False
>>> isinstance(obj2, MyClass2)
True
```

Esempio di classe in Python

- Il codice:

```
import math

class Cerchio:
    def __init__(self, c, r):
        self.centro = c
        self.raggio = r

    def area(self):
        return math.pi * self.raggio**2

c1 = Cerchio((0, 0), 5)
print "Il cerchio c1 ha raggio", c1.raggio
print "Il cerchio c1 ha centro", c1.centro
print "Il cerchio c1 ha area", c1.area()
print "c1 e' di tipo", type(c1)
```

- Produce come output:

```
Il cerchio c1 ha raggio 5
Il cerchio c1 ha centro (0, 0)
Il cerchio c1 ha area 78.5398163397
c1 e' di tipo <type 'instance'>
```

Ereditarietà

- Una classe può essere definita per derivazione da una classe base (*ereditarietà*)
- Una classe derivata può ridefinire (*overriding*) tutti i metodi di una classe base
- I tipi built-in del linguaggio non possono essere usati come classi base dal programmatore
- Per definire una classe `DerivedClass` come derivata da una classe `BaseClass` si usa la sintassi:

```
class DerivedClass(BaseClass):
    statement_1
    statement_2
    statement_3
    ...
```

- In un metodo `f` della classe derivata `DerivedClass` si può invocare il metodo omologo della classe base con il nome `BaseClass.f`
- Una classe può essere fatta derivare da una classe a sua volta derivata; in questo modo si possono realizzare *gerarchie di ereditarietà* multi-livello
- E' anche possibile realizzare l'*ereditarietà multipla* definendo una classe come `class DerivedClass(BaseClass1, BaseClass2, ...):`

Gerarchie di ereditarietà

- Una classe può essere fatta derivare da una classe a sua volta derivata
- In questo modo si possono realizzare *gerarchie di ereditarietà* multi-livello
- Una qualsiasi classe deriva dalla classe predefinita `object`
- Esempio:

```
class BaseClass:
    pass

class SecondLevelClass(BaseClass):
    pass

class ThirdLevelClass(SecondLevelClass):
    pass
```

- La funzione predefinita `issubclass(class1, class2)` restituisce:
  - `True` se `class1` è derivata da `class2` o da una sua sottoclasse
  - `False` altrimenti
- Con riferimento alle classi dell'esempio precedente:
  - `issubclass(SecondLevelClass, BaseClass)` restituisce `True`
  - `issubclass(ThirdLevelClass, BaseClass)` restituisce `True`
  - `issubclass(ThirdLevelClass, SecondLevelClass)` restituisce `True`