Introduzione al linguaggio Python

Prof.ssa Valeria Vittorini

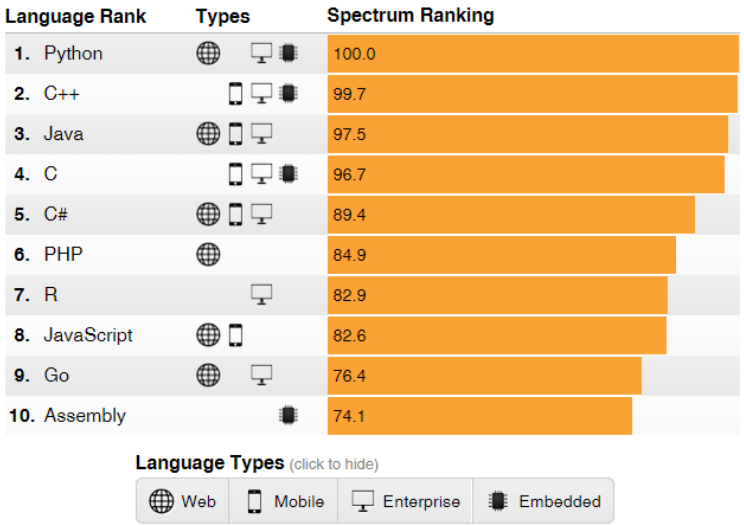
Prof. Roberto Canonico

Corso di Programmazione I

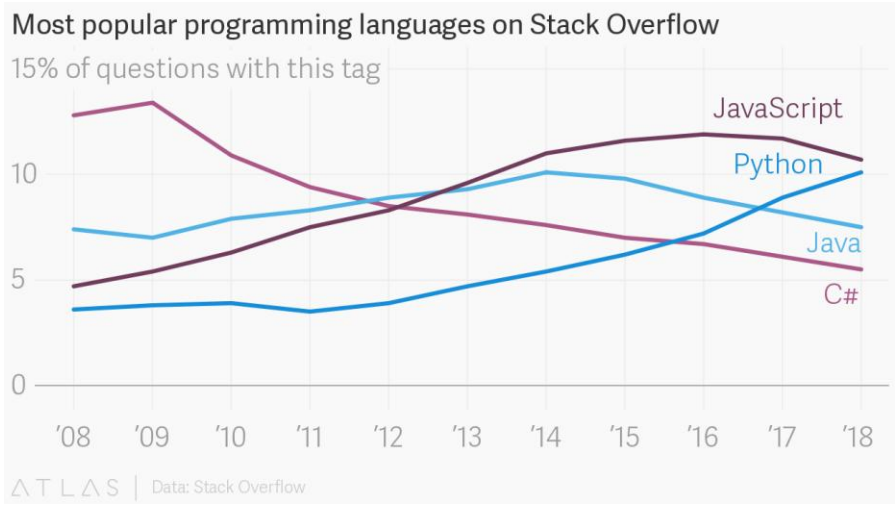
a.a. 2018-2019

Obiettivi

- Introdurre un linguaggio di programmazione la cui popolarità negli ultimi anni è in continua ascesa
- 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 2018



Fonte: https://www.theatlas.com/charts/BJlUylxqQ
su dati raccolti dallo
StackOverflow Developer Survey 2018
https://insights.stackoverflow.com/survey/2018/

Python: principali caratteristiche

- 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 e 3
  - Le due versioni non sono compatibili
  - 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
Python 3.4.4 (v3.4.4:737efcadf5a6, Dec 20 2015, 20:20:57) [MSC v.1600 64 bit (AMD64)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> 4+7
11
>>> print("Hello world!")
Hello world!
>>> exit()
C:\Users\Roberto>
```

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

Script Python

- Un programma Python è tipicamente scritto in un file sorgente testuale
- I file sorgente Python sono anche detti ***script*** perché possono essere eseguiti direttamente dall'interprete
- Per convenzione, gli script Python sono salvati in file con estensione **`.py`**
- Per lo sviluppo, una pratica comune è quella di usare un normale editor di testo per scrivere il codice sorgente

Esecuzione di script Python

- 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 queste lezioni useremo l'ambiente Thonny (https://thonny.org/)

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
```

- 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
a = 3 + 5 + 7 \ #continua nella prossima riga
+ 4 + 2 + 1
```

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
```

Sul concetto di oggetto in Python

- Un oggetto è un'entità caratterizzata da un insieme di attributi e metodi
- Attributi e metodi in Python sono generalmente pubblici
  - Si assumono noti i concetti della programmazione ad oggetti
- 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__', '__bool__', '__ceil__', '__class__', '__delatt
r__', '__dir__', '__divmod__', '__doc__', '__eq__', '__float__', '__floor__', '_
_floordiv__', '__format__', '__ge__', '__getattribute__', '__getnewargs__', '__g
t__', '__hash__', '__index__', '__init__', '__init_subclass__', '__int__', '__in
vert__', '__le__', '__lshift__', '__lt__', '__mod__', '__mul__', '__ne__', '__ne
g__', '__new__', '__or__', '__pos__', '__pow__', '__radd__', '__rand__', '__rdiv
mod__', '__reduce__', '__reduce_ex__', '__repr__', '__rfloordiv__', '__rlshift__
', '__rmod__', '__rmul__', '__ror__', '__round__', '__rpow__', '__rrshift__', '_
_rshift__', '__rsub__', '__rtruediv__', '__rxor__', '__setattr__', '__sizeof__',
 '__str__', '__sub__', '__subclasshook__', '__truediv__', '__trunc__', '__xor__'
, 'bit_length', 'conjugate', 'denominator', 'from_bytes', 'imag', 'numerator', '
real', 'to_bytes']
```

Esempi di invocazione di metodi su oggetti

- Sintassi:
  `nome_oggetto.nome_metodo(argomenti)`
- 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
```

Numeri in Python

- 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`** oppure **`long`**

  - Il tipo **`int`** è soggetto ad un limite di rappresentazione di macchina (es. 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`**

- 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`

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:
    - ' ' *stringa vuota*
    - () *tupla vuota*
    - [] *lista vuota*
    - {} *dizionario vuoto*

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

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))
<class 'str'>
>>> x = 123; print(type(x))
<class 'int'>
>>> x = 3.14; print(type(x))
<class 'float'>
>>> x = 3+5j; print(type(x))
<class 'complex'>
>>> x = [4, 5, 9]; print(type(x))
<class 'list'>
```

Sul concetto di variabile in Python (1)

- 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 Python, il modello di programmazione è significativamente diverso:**
  - il linguaggio effettua un controllo dinamico dei tipi (a tempo di esecuzione o *runtime*)
  - l'operatore di assegnazione semplicemente associa un nome ad un oggetto:

```
a = 5
```

- **a** è ***un nome*** per l'oggetto 5

Sul concetto di variabile in Python (2)

- Lo stesso "nome" può essere associato (*binding*) in momenti successivi ad oggetti diversi, anche di tipi diversi:

```
a = 5; a = "Pippo"
```

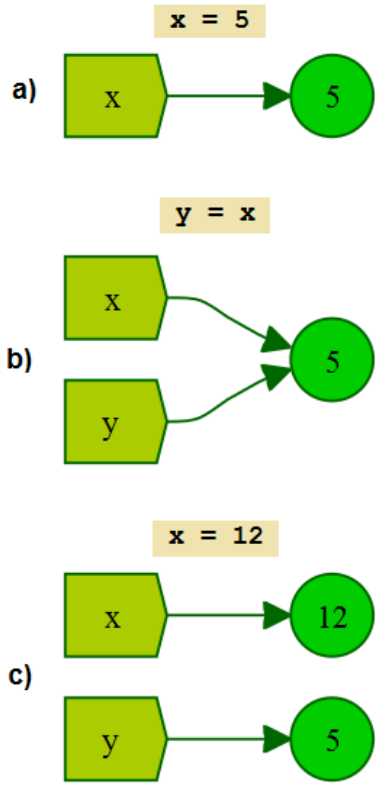
  - Questi **nomi** sono tipicamente chiamati **"variabili"** in Python

- **Un nome non può essere usato prima che sia stato associato ad un valore**
- **Un oggetto esiste fintanto che esiste un nome che fa riferimento ad esso**

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++***

La funzione `print` per l'output a video (Python 3)

- In Python 3 **`print()`** è una funzione che stampa sullo standard output in sequenza gli argomenti che le sono passati separati da uno spazio e termina andando a capo
- Possono essere specificati i parametri opzionali ***end*** e ***sep***
- Con il parametro opzionale ***end*** si specifica il carattere (o i caratteri) da appendere alla fine della riga. Il valore di default di *end* è *"*\n*"*
- Con il parametro opzionale ***sep*** si specifica il carattere (o i caratteri) che separano i vari argomenti della print. Il valore di default di *sep* è *" "* (1 spazio)

Esempio `print`

```
>>> print(1, 12, 99)
1 12 99
>>> print("A",end=""); print("B")
AB
>>> print("A",end=" "); print("B")
A B
>>> print("A",end="--"); print("B")
A--B
>>> print("A","B",sep="")
AB
>>> print("A","B",sep="-")
A-B
```

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
```

Funzioni per l'input da tastiera

- In Python 2:
  - la funzione **`raw_input()`** consente di leggere una stringa dallo standard input
  - la funzione **`input()`** consente di leggere un numero (`int` o `float`) in base 10 dallo standard input
    - se a runtime il valore fornito a **`input()`** non è un numero si genera un errore
- In Python 3
  - la funzione **`input()`** consente di leggere una stringa dallo standard input (tastiera)
  - Se si desidera acquisire da tastiera un numero intero occorre convertire la stringa ritornata da **`input()`** con la funzione di conversione **`int()`**
    - Se l'utente digita una stringa che non rappresenta un numero, si genera un errore

Esempi (Python2)

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

```
>>> 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
>>>
```

Esempi (Python3)

```
x = input("Digita una stringa e premi ENTER: ")
```

```
x = int(input("Digita un numero e premi ENTER: "))
```

```
>>> x = int(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>
ValueError: invalid literal for int()
with base 10: 'xxx'
>>>
```

Compound Statement

- I ***compound statement*** (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

```
#valore assoluto
a = int(input("Digita un numero intero e premi ENTER: "))
if (a >= 0):
    b = a
else:
    b = -a

print("b=", b)
```

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 = int(input("Digita un numero intero e premi ENTER: "))
# la funzione int puo' generare un errore
# se l'input da tastiera non e' corretto
# in questo caso il programma termina
print("Hai digitato",end=" ")
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`**

Operatori `and`, `or`, `not`

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

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

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

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` ⟶ `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 **`pass`**

- Lo statement **`pass`** rappresenta un'istruzione che non produce effetto
- Lo si usa a volte negli esempi di codice o quando si scrive un codice incompleto
- Esempio d'uso di **`pass`**:

```
if (a > 0):
    pass
b = 5
```

equivale al seguente statement C/C++

```
if (a > 0) {}  /* Se (a > 0) fai niente */
b = 5;         /* Istruzione successiva eseguita in qualsiasi caso */
```

Statement `while`

- **`while`** esegue un ciclo finché una condizione è vera

```
a = input("Digita qualsiasi cosa e poi ENTER (QUIT per terminare): ")
while (a != "QUIT"):
    print("Hai digitato: ", a)
    a = input("Digita qualsiasi cosa e poi ENTER (QUIT per terminare): ")

print("Hai digitato QUIT ed il programma termina.")
```

Statement **`while`** (2)

- Lo statement **`break`** nel corpo di un ciclo fa uscire dal ciclo
- Lo statement **`continue`** nel corpo di un ciclo salta all'iterazione successiva

```
conta = 0
while (True):
    a = int(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)
```

Statement `for`

- Lo statement **`for`** esegue una sequenza di istruzioni (corpo del ciclo) per tutti gli elementi di una struttura dati iterabile
  - Vedremo dopo le strutture dati in Python
- La sintassi è:

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

- Esempio:

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

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

Statement `for` (2)

- 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])
```

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 funzione_1():
    pass

def funzione_2(x):
    return x


ret = funzione_1()
print("Valore di ritorno =", ret)

ret = funzione_2(1)
print("Valore di ritorno =", ret)
ret = funzione_2("PIPPO")
print("Valore di ritorno =", ret)
```

```
Valore di ritorno = None
Valore di ritorno = 1
Valore di ritorno = PIPPO
```

Funzioni: esempio

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

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 = 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 globali

- Un nome di variabile usato in una funzione è trattato come una variabile locale
  - Il valore assegnato ad una variabile locale non si riflette sul valore assegnato ad una variabile con lo stesso nome definita all'esterno della funzione
- Gli argomenti di una funzione sono trattati alla stregua di variabili locali

```
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
```

Funzioni: argomenti, variabili locali e globali

- Per modificare in una funzione il valore associato ad un nome di variabile definito esternamente, si usa lo statement **`global`**

```
def f2(x):
    global a
    a = 3     # variabile globale
    x += 1    # argomento
    print("In f1:")
    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 valori di default

- Per ogni argomento di una funzione si può specificare un valore di default
  - Non è necessario che dato il default ad un argomento anche tutti gli argomenti che seguono abbiano un default (come invece in C++)
- 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

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

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],s[1],sep="")
AB
>>> 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],s[-2],sep="")
JI
>>> print(s[-10])
A
```

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'
>>>
```

Combinazione di stringhe e variabili

- 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 %5.3 f." % 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: 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'
>>>
```

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"`

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)`**

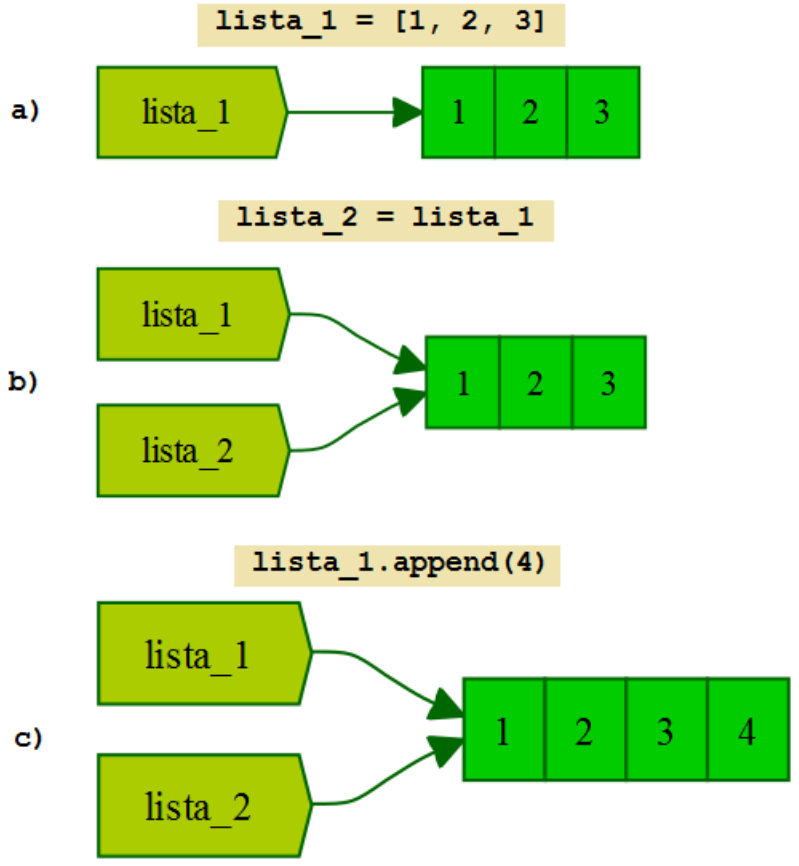
```
>>> l = [1, 2]
>>> m = ["Pippo", 3, 4]
>>> l + m
[1, 2, 'Pippo', 3, 4]
>>> l*4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> m[0]
Pippo
>>> m[0] = 99; print(m)
[99, 3, 4]
>>> len(m)
3
```

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

Liste: altre funzioni predefinite

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

NOTA: questa slide si riferisce a Python 2
In Python 3, range() restituisce un oggetto di classe "range" e non più una lista

```
>>> a = range(5)
>>> print(a)
range(0, 5)
>>> print(type(range(5)))
<class 'range'>
```

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 **`j=len(t)`**

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

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(0,10) ]
>>> print(l1)
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> 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"
>>> d["NA"]
'Naples'
>>> "NA" in d
True
>>> "AV" in d
True
>>> "MI" in d
False
```

Dizionari: metodi

Una lista 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`**

```
>>> 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()` accetta in ingresso una qualunque struttura dati iterabile e restituisce una lista di valori ordinata
  - E' opportuno non confondere `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. perché l'input fornito dall'utente non è corretto
- Con `try` è possibile "catturare" un evento di errore prodotto da uno statement
- Esempio (due versioni)

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

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

```
prompt  = "Inserisci un numero intero: "
warning = "Non hai inserito un numero \\
intero valido. Riprova."
while True:
    try:
        x = int(raw_input(prompt))
    except ValueError:
        print(warning)
        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 = 2018
def currentAge(birthYear):
    global currentYear
    return (currentYear-birthYear)

print("Libreria my_lib caricata")
```

```
# 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,end=" ")
   print(my_lib.currentAge(
      annoNascita_amici[amico]))
```

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

```
Libreria my_lib caricata
Eta' dei miei amici nell' anno 2018
Giovanni2: eta' 42
Mario: eta' 46
Andrea: eta' 40
Luigi: eta' 50
Giovanni: eta' 53
```

- 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.py
currentYear = 2018
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 2018
Giovanni2: eta' 42
Mario: eta' 46
Andrea: eta' 40
Luigi: eta' 50
Giovanni: eta' 53
```

```
# 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,end=" ")
   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

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, myClass)` restituisce `True` se `obj` è istanza della classe `myClass`, `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

```
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))
```

- L'output prodotto è:

```
Il cerchio c1 ha raggio 5
Il cerchio c1 ha centro (0, 0)
Il cerchio c1 ha area 78.53981633974483
c1 e' di tipo <class '__main__.Cerchio'>
```

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
...
```

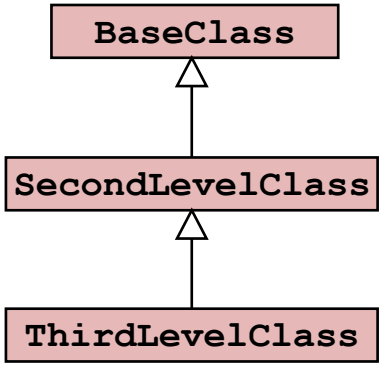
- In un metodo **`f`** della classe derivata **`DerivedClass`** si può invocare il metodo omologo della classe base con il nome **`BaseClass.f`**
- Una classe può essere fatta derivare da una classe a sua volta derivata
  - in questo modo si possono realizzare gerarchie di ereditarietà multi-livello
- 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`**