Introduzione al linguaggio Python

Prof.ssa Valeria Vittorini

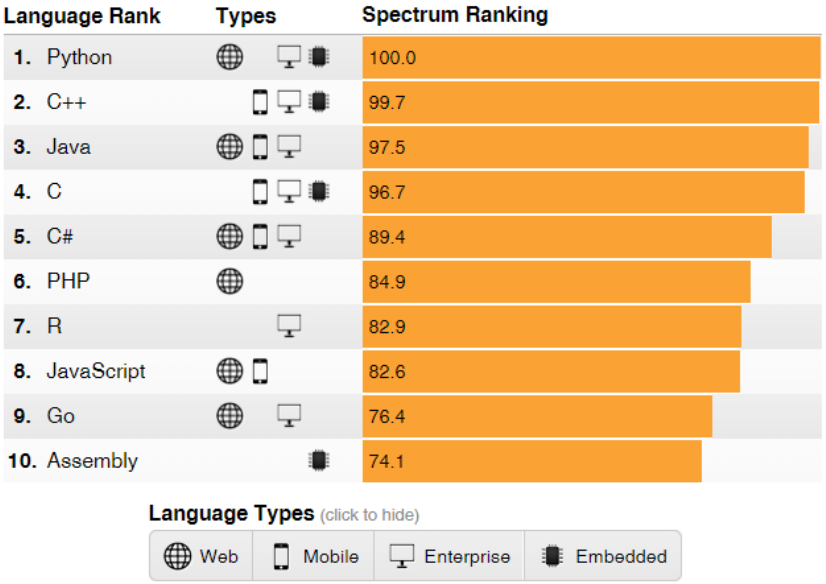
Prof. Roberto Canonico

Corso di Programmazione I

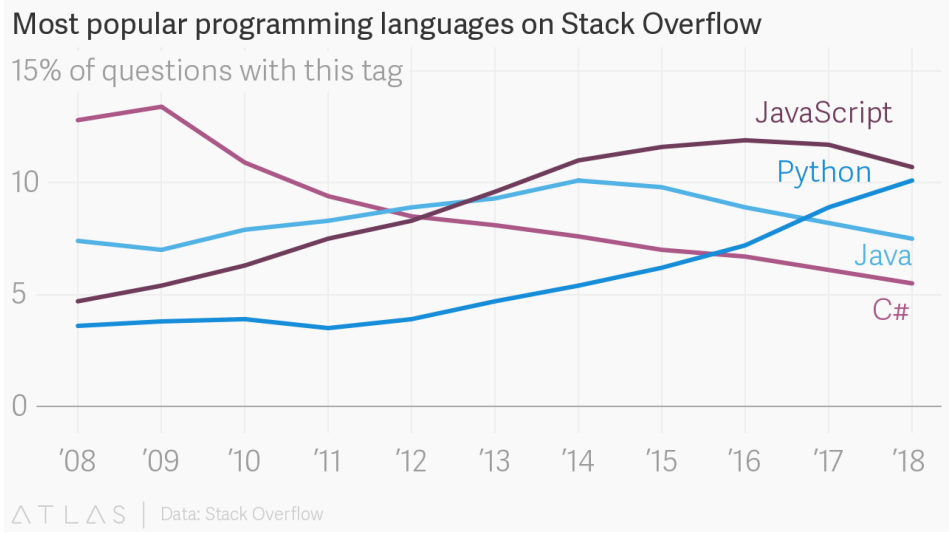
a.a. 2019-2020

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/

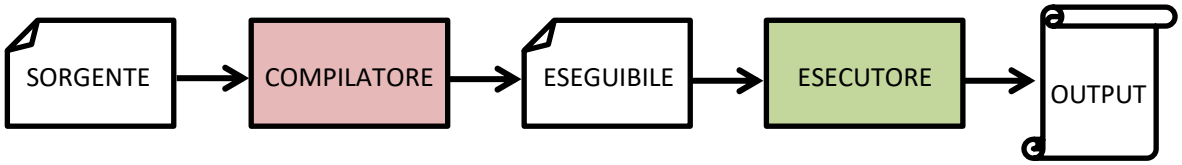
Python: principali caratteristiche

- 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/

Linguaggi interpretati e compilati

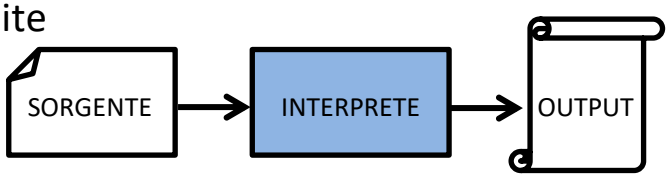
Linguaggi compilati: es. C, C++

- Il programmatore scrive il **codice sorgente** in un file di testo
- Il **compilatore** traduce il codice sorgente e produce un **codice eseguibile** (linguaggio macchina)
- L'esecutore carica il codice eseguibile nella memoria del computer e lo esegue



Linguaggi interpretati: es. Python

- Il programmatore scrive il **codice sorgente** in un file di testo
- L' **interprete** traduce (ad ogni esecuzione) il codice sorgente, trasformandolo in istruzioni del linguaggio macchina che vengono eseguite



Interprete Python: modalità interattiva

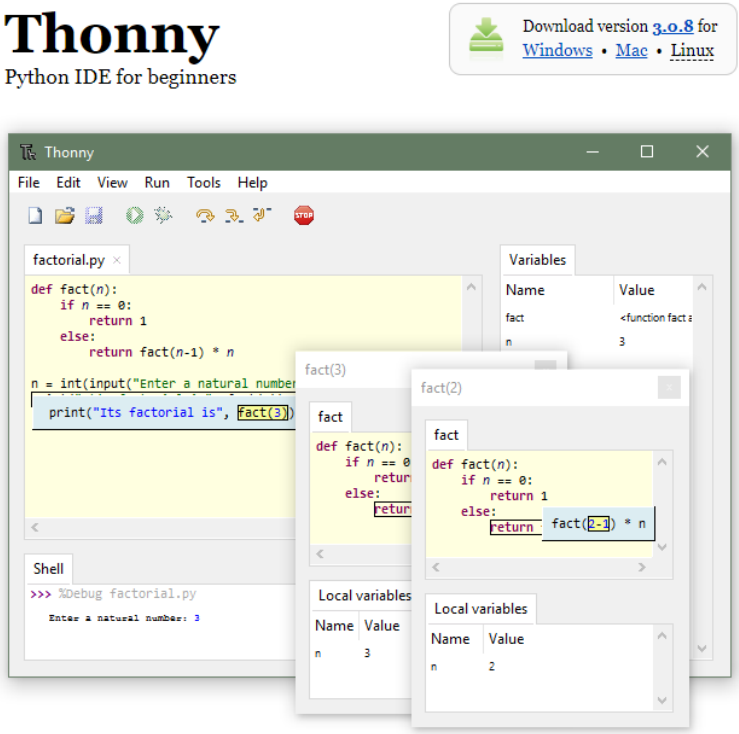
- Lanciando il programma eseguibile **`python`** si esegue l'interprete Python in ***modalità interattiva***
- Si ottiene un prompt dal quale si possono eseguire singoli statement Python
- In modalità interattiva i comandi vanno digitati da tastiera

```
C:\Users\Roberto>python
Python 3.7.0 (v3.7.0:1bf9cc5093, Jun 27 2018, 04:59:51) [MSC v.1914 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

L'ambiente di sviluppo Thonny

- Thonny è un semplice IDE per lo sviluppo di programmi Python
  - IDE = *Integrated Development Environment*
  - Installer di Thonny disponibile per Windows, MacOS e Linux su https://thonny.org
  - Sorgenti disponibili su GitHub
  - Thonny supporta Python 3 e non Python 2
  - Ha un interprete Python 3 distribuito con l'IDE ma può usare anche un altro interprete installato sulla macchina



Esecuzione di 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`**
- L'esecuzione del programma da una shell di comando del sistema operativo si ottiene avviando l'interprete Python e fornendo come input il *path* dello script
  - 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
    ```

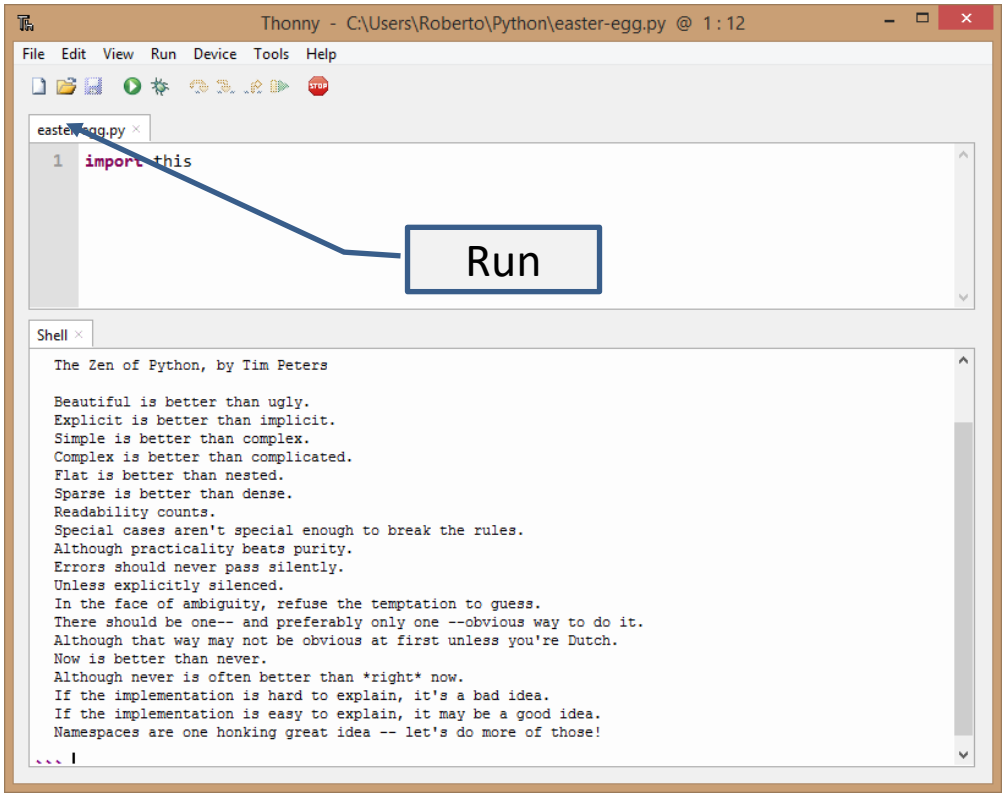
    è possibile eseguire direttamente lo script dal prompt dei comandi:

    ```
    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

Esecuzione di script Python in un IDE

- Gli ambienti di sviluppo consentono l'esecuzione diretta dello script dallo stesso ambiente di editing ed anche un supporto al debugging
  - Esecuzione dello script riga per riga
  - Ispezione dei valori delle variabili a run-time
  - …



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*

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, ogni statement deve avere la stessa indentazione del precedente, altrimenti si genera un errore in esecuzione***

error01.py

```
1  a = 1
2    b = 2
```

Shell

```
>>> %Run error01.py
  Traceback (most recent call last):
    File "c:\python\python37\lib\ast.py", line 35, in parse
      return compile(source, filename, mode, PyCF_ONLY_AST)
    File "C:\Users\Roberto\Python\error01.py", line 2
      b = 2
      ^
  IndentationError: unexpected indent

>>>
```

Compound statement ed indentazione

- 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

```
if (a >= 0):
    b = a
else:
    b = -a
```

Lo statement `print` (Python 2)

- 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
>>>
```

La funzione `print` (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
- 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"
  - Se *end*="" il successivo output sarà stampato sulla stessa riga senza spaziatura
- 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)
  - Se *sep*="" gli argomenti di print sono stampati sulla stessa riga senza spaziatura

```
>>> 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
```

Sul concetto di oggetto in Python

- Un oggetto è un'entità caratterizzata da un insieme di attributi e metodi
  - 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
```

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))
<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'>
```

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`**
- 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`

Operatori aritmetici

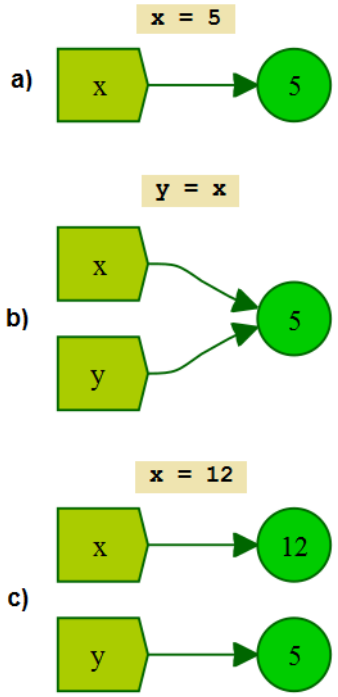
- In Python esistono i classici operatori aritmetici per le operazioni fondamentali:
  - `+` somma
  - `-` sottrazione
  - `*` prodotto
  - `/` divisione
  - `//` divisione intera
  - `%` resto modulo
  - `**` elevamento a potenza
- Non esistono gli operatori aritmetici unari ++ e -- del C/C++
- Se gli 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++***

Operatori 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: definizione ed invocazione

- Una funzione è un "pezzo di codice" (*sottoprogramma*) a cui il programmatore assegna un nome
- Con lo statement **`def nome_funzione():`**
  il programmatore “dà un nome” al codice delle funzione
- L’esecuzione del codice di una funzione avviene *invocando* la funzione mediante il suo nome (operazione di *function call*) seguito da parentesi ()

```
# Esempio di definizione di funzione
def saluto():
    print("Ciao, benvenuto!")


# Esempio di function call
saluto()
```

Funzioni: parametri

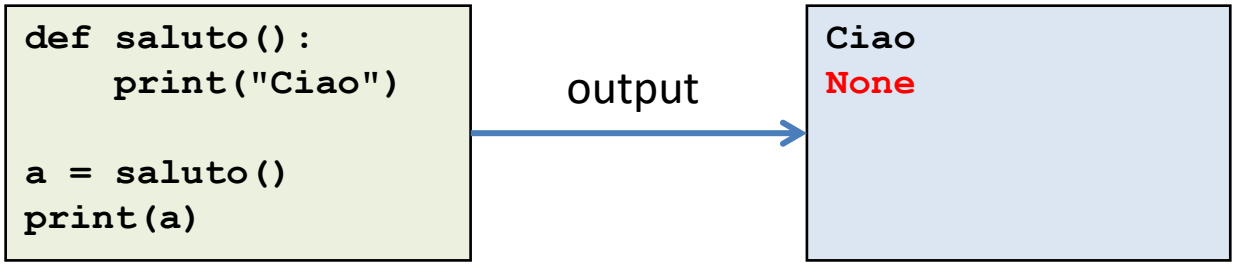
- Una funzione può ricevere in input una sequenza (tupla) di valori (*parametri*)
- Nella definizione di una funzione i parametri sono specificati tra parentesi dopo il nome
  - Es: `def max(x, y):`
  - Una funzione che non prevede parametri ha una lista di parametri vuota ()
- Quando una funzione viene invocata, i valori effettivi dei parametri sono specificati nella function call e possono essere o valori costanti o nomi di variabili
  - `max("Pippo", "Pluto")`
  - `max(a, b)`
- Attraverso l'uso delle funzioni con parametri, la stessa sequenza di codice può essere eseguita più volte su valori diversi

Funzioni: parametri

- Gli oggetti passati come parametri possono essere di qualsiasi tipo
  - Nessun controllo viene fatto dall'interprete sul tipo dei parametri passati alla funzione al momento della sua invocazione
  - Questo può essere un vantaggio …
  - ma anche fonte di possibili errori a tempo di esecuzione

Funzioni: valore di ritorno

- Lo statement **`return x`** inserito nella sequenza di istruzioni di una funzione, quando eseguito, termina l'esecuzione della funzione con la restituzione del valore x
- Al valore restituito con **`return`** da una funzione può essere assegnato un nome mediante uno statement di assegnazione
  - Esempio: **`m = max(a, b)`**
- 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



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

- 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 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: esempio d'uso di `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 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)
```

side effect

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

Ancora su variabili locali e globali (1)

- Se una funzione fa riferimento ad una variabile definita all'esterno, senza tentarne di modificarne il valore, l'interprete assume che il nome usato nella funzione sia quello della variabile globale

```
def f():
    print(s) # variabile globale
s = "I love Paris in the summer!"
f()
```

```
I love Paris in the summer!
```

- Se in una funzione si fa uso di un nome di variabile in una espressione di assegnazione, l'interprete assume che il nome identifichi una variabile locale distinta da una eventuale variabile globale con lo stesso nome

```
def f():
    s = "I love London!"
    print(s) # variabile locale
s = "I love Paris!"
f()
print(s)
```

```
I love London!
I love Paris!
```

Ancora su variabili locali e globali (2)

- Se in una funzione si usa un nome di variabile in un'espressione di assegnazione, l'interprete assume che il nome identifichi una variabile locale distinta da una eventuale variabile globale omonima

```
def f():
    print(s)    # variabile locale
    # ERRORE: nome usato prima che
    # sia stato assegnato un valore
    s = "I love London!"
    print(s)
s = "I love Paris!"
f(); print(s)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in f
UnboundLocalError: local variable 's'
referenced before assignment
```

- Lo statement `global s` in una funzione istruisce l'interprete che il nome s si riferisce ad una variabile globale definita all'esterno della funzione

```
def f():
    global s
    print(s)    # variabile globale
    s = "I love London!"
    print(s)    # variabile globale
s = "I love Paris in the summer!"
f(); print(s)
```

```
I love Paris in the summer!
I love London!
I love London!
```

Funzioni: argomenti con valori 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:",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
```

- 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

- In Python 3
  - la funzione **`input()`** consente di leggere una stringa dallo standard input (tastiera)

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

- 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

```
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
>>>
```

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'>
```

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 `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`**
- Lo statement **`pass`** rappresenta un'istruzione che non produce effetto
- 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 premi ENTER (QUIT per terminare): ")
while (a != "QUIT"):
    print("Hai digitato: ", a)
    a = input("Digita qualsiasi cosa e premi ENTER (QUIT per terminare): ")

print("Hai digitato QUIT 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 = 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)
```

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],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
```

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: *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)`**

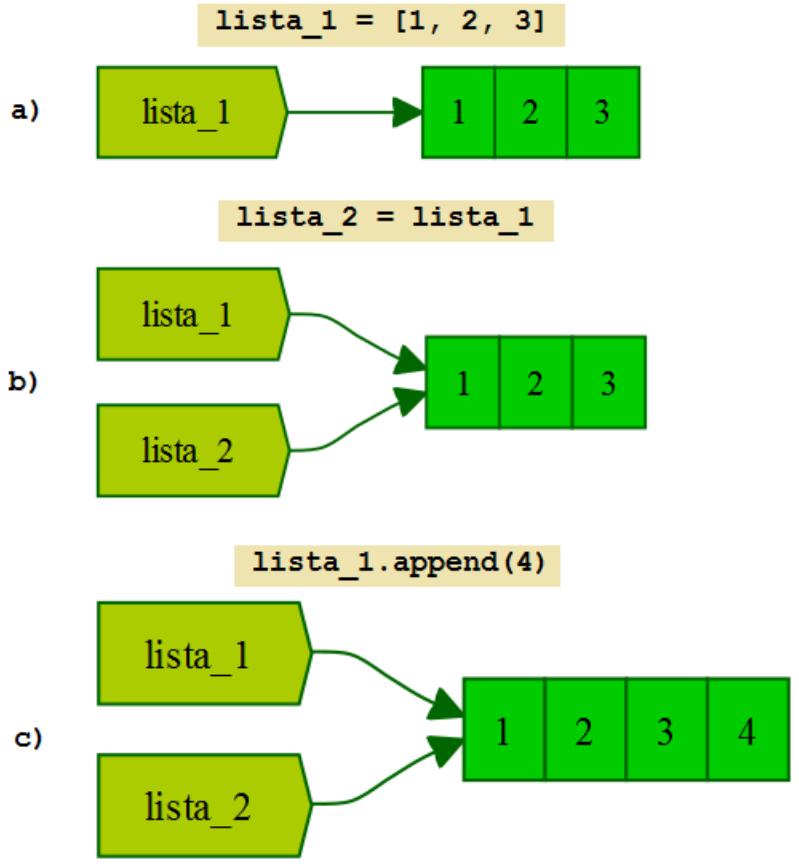
```
>>> l = [1, 2]
>>> m = ["Pippo", 3, 4]
>>> l + m
[1, 2, 'Pippo', 3, 4]
>>> l*4
[1, 2, 1, 2, 1, 2, 1, 2]
>>> m[0]
Pippo
>>> m[0] = 99; print(m)
[99, 3, 4]
>>> len(m)
3
```

```
>>> 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]
()
```

Statement `for`

- Lo statement **`for`** esegue una sequenza di istruzioni (corpo del ciclo) per tutti gli elementi di una struttura dati iterabile
- La sintassi 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(0,10) ]
>>> 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']
```

La clausola `else` nei cicli `while` e `for`

- Gli statement **`while`** e **`for`** prevedono una clausola opzionale **`else`** che introduce una sequenza di istruzioni (*clausola di uscita*) che viene eseguita una volta sola all'uscita dal ciclo
- Se l'uscita dal ciclo avviene con l'istruzione **`break`**, la clausola di uscita ***non viene eseguita***

```
while cond:
    corpo_del_ciclo
else:
    clausola_di_uscita
statement_successivo
```

```
for iterating_var in sequence:
    corpo_del_ciclo
else:
    clausola_di_uscita
statement_successivo
```

Esempio d'uso della clausola `else` in `for`

- Un esempio in cui la clausola `else` può essere utile:
  - un ciclo `for` dal quale ci si aspetta di terminare con `break` e la scansione dell'intera sequenza iterabile rappresenta una situazione anomala

```
for element in values:
   if element == 5:
        print("Valore 5 trovato")
        break
else:
    print("Valore 5 non trovato")
```

Un altro esempio d'uso della clausola `else`

- Un programma (con due cicli `for` innestati) per verificare se una lista contenga solo numeri primi
- Se si verifica una certa condizione nel ciclo interno, si vuole uscire anche dal ciclo più esterno

```
for k in [2, 3, 5, 7, 8, 11]:
   print("Verifico se %d è primo" % k)
   for m in range(2, k):
      if (k % m == 0):
         print("%d è divisibile per %d"%(k, m))
         break
   else:
      print("Non sono stati trovati divisori: %d è PRIMO" % k)
      continue
   print("ERRORE: la lista contiene il numero %d che non è PRIMO" % k)
   break
else:
    print("OK: la lista non contiene numeri NON primi")
```

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'}
```

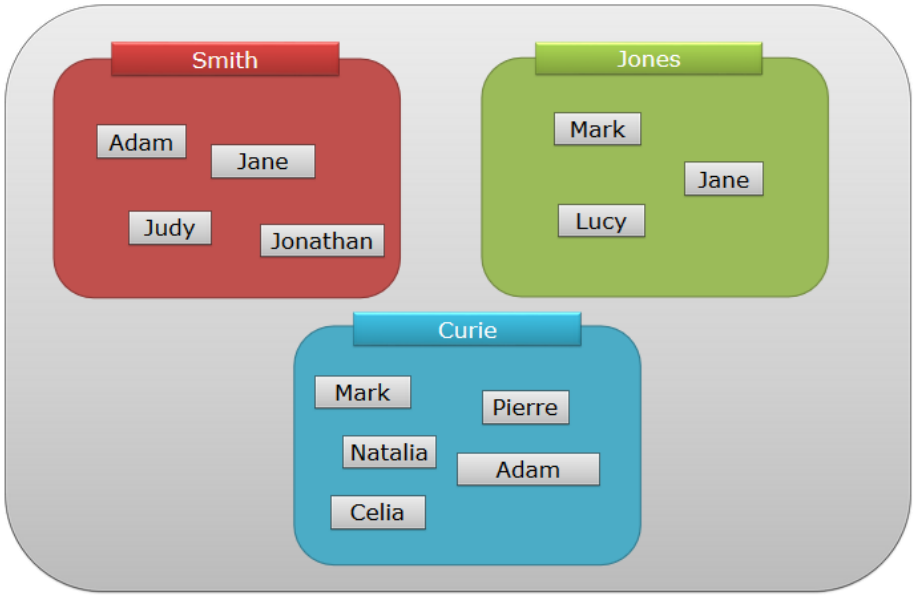
La funzione predefinita `sorted`

- 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]
>>>
```

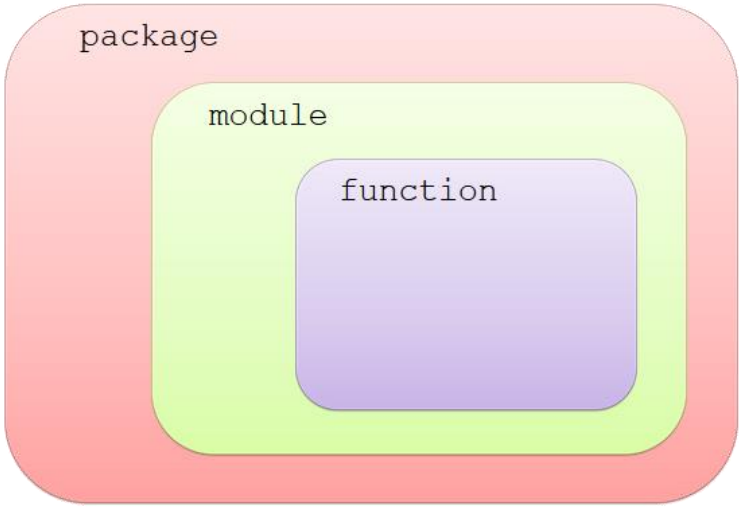
Il concetto di namespace

- Un ***namespace*** è un contesto logico nel quale ciascun identificatore è unico
  - Ad esempio, in uno script non possono esistere due variabili globali con lo stesso nome
- In namespace differenti possono esistere oggetti differenti con lo stesso nome
  - All'interno di una funzione può esistere una variabile locale con lo stesso nome di una variabile globale: una funzione ha un suo proprio namespace locale



Moduli e package in python

- Un ***modulo*** è un file nel quale si trovano funzioni e statement Python
- Esistono sia moduli distribuiti insieme all'interprete (*built-in*) che moduli distribuiti separatamente e resi disponibili all'interprete mediante un tool di installazione
- Un ***package*** è una collezione di moduli i cui file sono collocati nella stessa directory



Uso di librerie in Python

- 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 moduli in Python: `import`

- Per rendere disponibile all'interprete Python il codice di un modulo si usa lo statement **`import`**
- Lo statement:

  `import lib_XX`

  rende «visibili» all'interprete tutti i nomi (variabili e funzioni) definiti a livello globale nel file **`lib_XX.py`**
  - I nomi di variabili e funzioni della libreria sono associati al ***namespace*** **`lib_XX`**
  - Il nome **`name`** definito in **`lib_XX.py`** dovrà essere riferito come **`lib_XX.name`** in uno script che fa **`import lib_XX`**
  - Se lo script **`lib_XX.py`** contiene statement eseguibili, essi sono eseguiti nel momento in cui è eseguito l'import
  - Il file **`lib_XX.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:\Python37\Lib\site-packages`**)

Il modulo `math`

- Il modulo **`math`** contiene variabili e funzioni significative per il calcolo matematico
- Se un programma contiene **`import math`**, il programmatore potrà usare, ad esempio:
  - L'oggetto numerico float **`math.pi`** che ha il valore di π
  - Le funzioni trigonometriche **`math.sin()`**, **`math.cos()`**, **`math.tan()`** e le rispettive inverse **`math.asin()`**, **`math.acos()`**, **`math.atan()`**
  - Le funzioni **`math.exp()`**, **`math.log()`**, …
  - e molte altre …

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

`import` di moduli con alias

- Lo statement:

  `import lib_XX as lib_alt`

  inserisce nel namespace `lib_alt` tutti i nomi presenti in `lib_XX.py`
  - Il nome `name` definito in `lib_XX.py` dovrà essere riferito come `lib_alt.name`
    in uno script che fa `import lib_XX`
- Esempio:

```
import math as M
print(M.sin(M.pi/2))
```

Uso di librerie in Python: `from-import`

- Lo statement

```
from lib_XX import name
```

  rende visibile all'interprete nel namespace principale dello script corrente il nome **`name`** definito nella libreria **`lib_XX.py`**
  - In questo caso, il nome **`name`** definito in **`lib_XX.py`** potrà essere direttamente riferito come **`name`** nello script che fa l'import
- Lo statement

```
from lib_XX import *
```

  rende visibili all'interprete nel namespace dello script corrente tutti i nomi definiti nella libreria **`lib_XX.py`**
  - Anche in questo caso, un nome **`name`** definito in **`lib_XX.py`** potrà essere direttamente riferito come **`name`** nello script che fa l'import
- **Occorre fare attenzione a possibili conflitti di nomi definiti in script differenti**

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

Alias di identificatori con `from-import-as`

- Lo statement:

  `from lib_XX import name as alt_name`

  inserisce nel namespace principale dello script corrente il nome **`name`** definito in **`lib_XX.py`** cambiandone il nome in **`alt_name`**
- Esempio:

```
from math import sin as sen, pi as pigreco
print(sen(pigreco/2))
```

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
```

Esempi di gestione delle eccezioni

- Due esempi equivalenti

```
prompt  = "Inserisci un numero intero: "
warning = "Input non 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 = "Input non valido. Riprova."
while True:
    try:
        x = int(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
```

Esempi di gestione delle eccezioni (2)

```
prompt  = "Inserisci un numero intero diverso da zero: "
warning = "Non hai inserito un numero intero valido. Riprova."
while True:
    try:
        x = int(input(prompt))
        y = 1/x
        break
    except:
        print(warning)
print("Reciproco di", x, "=", y)
```

- Esempio di esecuzione:

```
Inserisci un numero intero diverso da zero: 0
Non hai inserito un numero intero valido. Riprova.
Inserisci un numero intero diverso da zero: a
Non hai inserito un numero intero valido. Riprova.
Inserisci un numero intero diverso da zero: 2
Reciproco di 2 = 0.5
```

Esempi di gestione delle eccezioni (3)

```
prompt  = "Inserisci un numero intero diverso da zero: "
warning_1 = "Non hai inserito un numero intero valido. Riprova."
warning_2 = "Hai inserito zero. Riprova."
while True:
    try:
        x = int(input(prompt))
        y = 1/x
        break
    except ValueError:
        print(warning_1)
    except ZeroDivisionError:
        print(warning_2)
print("Reciproco di", x, "=", y)
```

- Esempio di esecuzione:

```
Inserisci un numero intero diverso da zero: a
Non hai inserito un numero intero valido. Riprova.
Inserisci un numero intero diverso da zero: 0
Hai inserito zero. Riprova.
Inserisci un numero intero diverso da zero: 2
Reciproco di 2 = 0.5
```

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: esercizio sui numeri primi

- Problema: determinare il numero di numeri primi presenti in ciascuna centinaia fino a 1000
- Esercizio: si estenda il codice dell'esempio precedente, producendo una lista in cui
  - il primo elemento sia il numero di numeri primi compresi tra 0 e 99 cioè 25
  - il secondo elemento sia il numero di numeri primi compresi tra 100 e 199 cioè 21
  - ....
- Suggerimento:
  - si deve produrre e stampare una lista **`n`** di 10 elementi, in cui **`n[0]`** dovrà contenere il numero di numeri primi compresi tra 0 e 99, e così via
  - si costruisca la lista **`p`** di tutti i numeri primi fino a 1000 usando la funzione **`primes(1000)`**, dopodiché si esegua un ciclo di scansione degli elementi di questa lista **`p`** e, per ogni elemento, se ne determini la centinaia di appartenenza e si aggiorni il corrispondente elemento in **`n`**, incrementandolo di 1
  - Infine, si stampi il vettore **`n`**

Esempio #1-bis: esercizio sui numeri primi (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: 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 da file CSV (1)

- Scrivere un programma Python che estragga da un file dati testuale in formato CSV (*comma separated values*) 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 da file CSV (2)

- Soluzione (prima parte):

```
# File: elabora-voti.py
csv_file = open("voti.csv", "r")
lines = csv_file.readlines()[1:] # Ignora la prima linea del file
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()
....
```

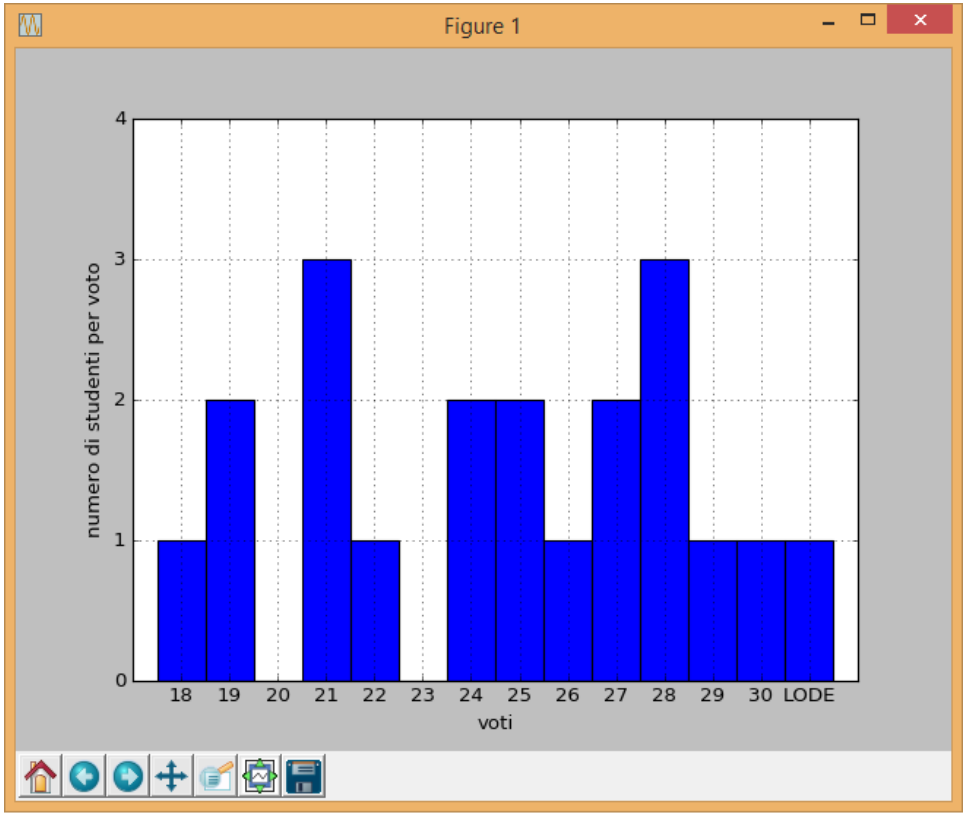
Esempio #3: analisi di dati da file CSV (3)

- Soluzione (seconda parte):

```
# 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 da file CSV (4)

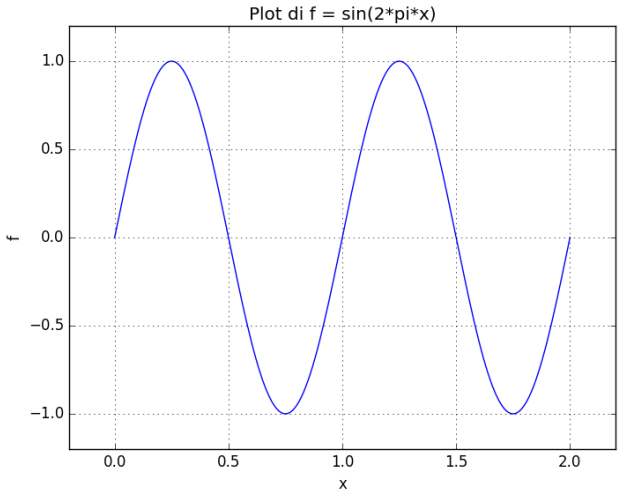
- Il programma produce come output la figura seguente:



Esempio #4: plot di una funzione

```
import numpy as np
import matplotlib.pyplot as plt

x = np.arange(0.0, 2.01, 0.01)
y = np.sin(2*np.pi*x)

plt.plot(x, y)
plt.margins(0.1)

plt.xlabel('x')
plt.ylabel('f')
plt.title('Plot di f = sin(2*pi*x)')
plt.grid(True)
plt.savefig("test.png")
plt.show()
```

- Output prodotto (**`test.png`**):



Esempio #4: plot di una funzione

```
x = np.arange(0.0, 2.01, 0.01)
y = np.sin(2*np.pi*x)
```

- **`np.arange(start,end,step)`** restituisce una lista di numeri **`float`** prodotti a partire da **`start`** con incremento **`step`** fino al valore massimo **`end`** (escluso)

```
[ 0.    0.01  0.02  0.03  0.04  0.05  0.06  0.07  0.08  0.09  0.1   0.11
  0.12  0.13  0.14  0.15  0.16  0.17  0.18  0.19  0.2   0.21  0.22  0.23
  0.24  0.25  0.26  0.27  0.28  0.29  0.3   0.31  0.32  0.33  0.34  0.35
  0.36  0.37  0.38  0.39  0.4   0.41  0.42  0.43  0.44  0.45  0.46  0.47
  0.48  0.49  0.5   0.51  0.52  0.53  0.54  0.55  0.56  0.57  0.58  0.59
  0.6   0.61  0.62  0.63  0.64  0.65  0.66  0.67  0.68  0.69  0.7   0.71
  0.72  0.73  0.74  0.75  0.76  0.77  0.78  0.79  0.8   0.81  0.82  0.83
  0.84  0.85  0.86  0.87  0.88  0.89  0.9   0.91  0.92  0.93  0.94  0.95
  0.96  0.97  0.98  0.99  1.    1.01  1.02  1.03  1.04  1.05  1.06  1.07
  1.08  1.09  1.1   1.11  1.12  1.13  1.14  1.15  1.16  1.17  1.18  1.19
  1.2   1.21  1.22  1.23  1.24  1.25  1.26  1.27  1.28  1.29  1.3   1.31
  1.32  1.33  1.34  1.35  1.36  1.37  1.38  1.39  1.4   1.41  1.42  1.43
  1.44  1.45  1.46  1.47  1.48  1.49  1.5   1.51  1.52  1.53  1.54  1.55
  1.56  1.57  1.58  1.59  1.6   1.61  1.62  1.63  1.64  1.65  1.66  1.67
  1.68  1.69  1.7   1.71  1.72  1.73  1.74  1.75  1.76  1.77  1.78  1.79
  1.8   1.81  1.82  1.83  1.84  1.85  1.86  1.87  1.88  1.89  1.9   1.91
  1.92  1.93  1.94  1.95  1.96  1.97  1.98  1.99  2.  ]
```

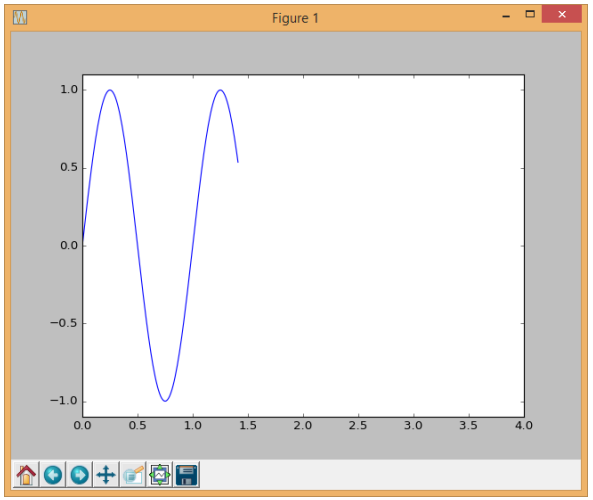
- **`np.sin(2*np.pi*x)`** (essendo **`x`** una lista) restituisce una lista di numeri **`float`** prodotti applicando la funzione sin(2*np.pi*x) a ciascun elemento di x

```
[  0.00000000e+00   6.27905195e-02   1.25333234e-01   1.87381315e-01
   2.48689887e-01   3.09016994e-01   3.68124553e-01   4.25779292e-01
   4.81753674e-01   5.35826795e-01   5.87785252e-01   6.37423990e-01
   6.84547106e-01   7.28968627e-01   7.70513243e-01   8.09016994e-01
   8.44327926e-01   8.76306680e-01   9.04827052e-01   9.29776486e-01
   9.51056516e-01   9.68583161e-01   9.82287251e-01   9.92114701e-01
...
```

Esempio #5: plot animato di una funzione

```
import math
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

x = []
y = []
x0 = 0.0
delta = 0.01

fig = plt.figure()
ax = plt.axes()
ax.set_ylim(-1.1, 1.1)
ax.set_xlim(0, 4)
lineS, = ax.plot(x, y)

def init():
    lineS.set_data([], [])
    return lineS,

def animate(i):
    current_x = x0+delta*i
    current_y = math.sin(2*math.pi*current_x)
    x.append(current_x)
    y.append(current_y)
    lineS.set_data(x, y)
    return lineS,

anim = animation.FuncAnimation(fig, animate, interval=100)
plt.show()
```

- Output prodotto:



Esempio #6: reperimento di un file dal web

```
import urllib.request
import os

filename = "Elenco-comuni-italiani.csv"
url = "https://www.istat.it/storage/codici-unita-amministrative/Elenco-comuni-italiani.csv"

if (not os.path.isfile(filename)):
    print("Il file", filename, "non e' presente. Lo scarico dal sito ISTAT.")
    urllib.request.urlretrieve(url, filename))
    print("Il file", filename, "e' stato scaricato dal sito ISTAT.")
else:
    print("Il file", filename, "e' gia' disponibile localmente.")
```

Esempio #7: analisi di un file CSV

```
import urllib
import os

...


csv_file = open(filename, "r", encoding="latin-1")
lines = csv_file.readlines()[1:] # Ignora la prima linea del file
comuni = {}      # Si usa un dizionario per mantenere le coppie comune:codice_catastale
i = 1
for line in lines:
    line = line.rstrip('\n')
    data_item = line.split(';')
    comune = data_item[5]
    codice_catastale = data_item[18]
    if (comune == ""):
        continue
    comuni.update({comune : codice_catastale})

csv_file.close()

while (1):
    comune = input("Inserisci un comune italiano: ")
    codice = comuni.get(comune)
    if (codice != None):
        print("Codice catastale di", comune, "=", codice)
    else:
        print("Comune non esistente in Italia")
```

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 78.5398163397
c1 e' di tipo <class '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
...
```

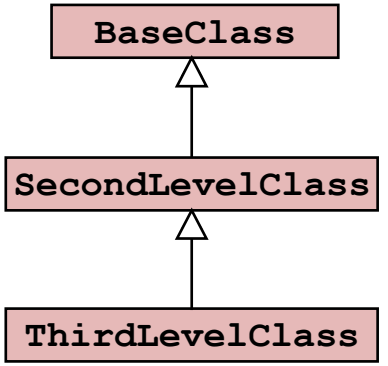
- In un metodo **`f`** della classe derivata **`DerivedClass`** si può invocare il metodo omologo della classe base con il nome **`BaseClass.f`**
- Una classe può essere fatta derivare da una classe a sua volta derivata
  - in questo modo si possono realizzare gerarchie di ereditarietà multi-livello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