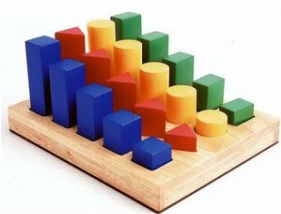


Python per programmatori

Vincenzo Bonifaci

Università degli Studi Roma Tre



Generalità su Python

Perché Python?

Python è un linguaggio di programmazione creato nel 1991 da Guido van Rossum (Olanda)



- ▶ Linguaggio imperativo e ad alto livello
- ▶ Orientato a leggibilità ed immediatezza d'uso
- ▶ Concezione moderna
- ▶ Relativamente consolidato

Python versione 3: lanciato nel 2008

Una buona fetta del vocabolario Python

tipi int float str bool list range	punteggiatura () [] { } : ,	relazioni == != < <= > >= in	operazioni numeriche + - * / // %	operazioni su stringhe + % ord() chr() .split() .join()	operazioni booleane True False not and or
input/output input() print() sys.stdin sys.stdout	controllo di flusso if else for while return	sequenze (...) [...] len()	orientamento agli oggetti class self __init__	funzioni matematiche min() max() abs() pow()	
assegnazione =	spazio dei nomi def import from __name__	gestione errori raise try except assert			

Installare python sul proprio sistema

- ▶ Scaricabile da www.python.org/downloads/
- ▶ Già installato su molte versioni di Mac OS X e Linux, ma può essere una versione obsoleta
 - ▶ In alcuni casi il comando `python` richiama la versione 2, `python3` la versione 3
- ▶ Queste slide: Python 3.5 o versione successiva
- ▶ Verifica della versione:

```
vincenzo@euler:~# python --version
Python 2.7.12 # antiquato...
vincenzo@euler:~# python3 --version
Python 3.5.2 # OK!
```

Impostazione del PATH

- ▶ Per poter invocare `python` da qualunque cartella è necessario che la cartella in cui è installato `python` sia nell'elenco delle cartelle ricercate dal sistema operativo, detto PATH
- ▶ **Assicurarsi** che il PATH venga aggiornato dal programma di installazione di Python!

Impostazione del PATH



La documentazione di Python

- ▶ Le funzionalità di Python sono dettagliate nella **documentazione online**, all'indirizzo
docs.python.org
- ▶ **Language reference**: descrive la sintassi e la semantica del linguaggio Python
- ▶ **Library reference**: descrive le tante *funzioni* predefinite:
 - ▶ Le funzioni sono organizzate in “*moduli*”, ciascuno dedicato ad un differente dominio (matematica, stringhe, sistema operativo, connessioni di rete...)

L'interprete interattivo (console)

```
vincenzo@euler:~# python3
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> 2 + 3
5
>>> 2 ** 3
8
>>> 2 / 3
0.6666666666666666
>>> def f(x): return x*x*x
...
>>> f(10)
1000
>>> ^D
```

Per abbandonare l'interprete:

Ctrl + Z su Windows, Ctrl + D su Linux/Mac

Sviluppo di programmi Python

- ▶ Un *programma* Python è una sequenza di *istruzioni*
- ▶ Memorizzato in un *file* di testo con estensione **.py**

Esempio di programma: hello.py

```
1  # Programma di benvenuto
2  name = input("Come ti chiami? ")
3  print("Buonasera, " + name + "!")
```

Questo programma è composto da tre righe:

1. Un *commento*, ignorato da Python
2. Istruzione che legge una riga di caratteri dal canale di ingresso e li memorizza in una *variabile* chiamata `name`
3. Istruzione che stampa in uscita "Buonasera, ", seguito dai caratteri memorizzati nella variabile `name`, seguito da "!"

Esecuzione in modalità *script*

- ▶ Alternativa all'uso dell'interprete interattivo
- ▶ Esegue le istruzioni contenute in un file `.py`

```
~# python3 hello.py  
Vincenzo  
Ciao, Vincenzo!
```

Variabili, tipi di dati ed espressioni

Variabili e valori

- ▶ Una *variabile* non è altro che un **nome**, a cui è associato un *valore* numerico, testuale, o di altro tipo:

```
>>> a = 10 # assegna il valore 10 alla variabile 'a'  
>>> print(a)  
10
```

- ▶ Un'istruzione di *assegnazione* della forma

var = V

associa il valore V alla variabile var

```
>>> a = 7 # assegna il valore 7 alla variabile 'a'  
>>> print(a)  
7
```

Variabili e valori (segue)

- ▶ I valori sono di vario *tipo*:
 - ▶ *numero intero* (`int`): 4, -3, ...
 - ▶ *numero decimale* (`float`): 11.562, -2.17, ...
 - ▶ *stringa* (`str`): `'abc'`, `"questa è una stringa"`, ...
 - ▶ *booleano* (`bool`): `True`, `False`
 - ▶ *lista* (`list`): `[1, 5, 7]`, `['a', 'bc']`, ...
 - ▶ ...
- ▶ Le variabili, di per sé, **non hanno** tipo in Python: possono memorizzare qualunque tipo di valore
- ▶ Python è un linguaggio *tipato dinamicamente*, a differenza di altri *tipati staticamente* quali C o Java

Variabili e valori (segue)

Per conoscere il tipo di un valore si può usare la funzione predefinita `type()`

```
>>> a = "Ciao"  
>>> b = 17  
>>> type(a)  
<class 'str'>  
>>> type(b)  
<class 'int'>  
>>> type(type(b))  
<class 'type'>
```


Conversione di tipo

```
>>> int(3.9) # le cifre dopo la virgola vengono scartate
3
>>> int('12') # la stringa viene interpretata come intero
12
>>> float(17) # intero "promosso" a decimale
17.0
>>> str(12) # l'intero viene convertito in stringa
'12'
```

Identificatori di variabile

- ▶ Un nome di variabile può essere arbitrariamente lungo ed essere composto da lettere, cifre e dal carattere *underscore* (`_`)
- ▶ Non può **iniziare** con una cifra
- ▶ Non può essere una delle **parole chiave riservate** del linguaggio (`True`, `False`, `int`, `if`, `for`, ...)

Espressioni

- ▶ Combinando valori, variabili, operatori e funzioni si ottiene un'*espressione*
- ▶ L'interprete interattivo *valuta* ogni espressione immessa e ne stampa il valore:

```
>>> 1 + 1
2
>>> True and False
False
>>> type(True)
<class 'bool'>
```

- ▶ L'esecuzione in modalità *script* invece, valuta, **ma non stampa**, i valori delle espressioni

Numeri interi (`int`)

Gli interi sono codificati in Python con valori di tipo `int`:

```
>>> type(-3)
<class 'int'>
>>> type(0)
<class 'int'>
>>> type(123412341234)
<class 'int'>
```

Diversamente da altri linguaggi, **non c'è un limite a priori** alla magnitudine degli interi rappresentati

Numeri decimali (float)

I numeri reali sono rappresentati come sequenze **finite** di cifre prima e dopo la “virgola” (il punto) decimale:

```
>>> type(12.5)
<class 'float'>
>>> -12.5 + 1.7
-10.8
>>> 23.1 * -2
-46.2
>>> type(-4)
<class 'int'>
>>> type(-4.0)
<class 'float'>
```

Numeri come π , $\sqrt{2}$ o anche $1/3$ sono rappresentabili solo in maniera **approssimata** da valori **float**

IEEE-754: 52 bit di mantissa, 11 bit di esponente, 1 bit di segno

Operazioni numeriche in Python 3

- ▶ Addizione: $2 + 3 \rightarrow 5$
- ▶ Sottrazione: $5 - 2 \rightarrow 3$
- ▶ Moltiplicazione: $3 * 4 \rightarrow 12$
- ▶ Divisione: $17 / 5 \rightarrow 3.4$
- ▶ Divisione intera: $17 // 5 \rightarrow 3$
- ▶ Resto della divisione: $17 \% 5 \rightarrow 2$
- ▶ Elevamento a potenza: $2 ** 3 \rightarrow 8$

Nota: da Python 2 a Python 3 le convenzioni sulla divisione sono cambiate!

Operazioni tra `int` e `float`

Le operazioni miste tra `int` e `float` sono operate:

1. convertendo l'operando intero a `float`
2. eseguendo l'operazione

Il risultato è **in ogni caso** di tipo `float`

```
>>> 15.7 + 3
18.7
>>> 18.0 * 5
90.0
```

Anche l'operatore `"/` restituisce sempre un `float`

```
>>> 10 / 5 # risultato è decimale anche se la divisione è esatta
2.0
>>> 10 // 5 # usiamo // se desideriamo la divisione intera
2
```

Stringhe (`str`)

Una *stringa* è una **sequenza di caratteri**

- ▶ Delimitata da apici singoli (`'abc'`) o doppi (`"xyz"`)

Operazioni basilari:

- ▶ Concatenazione: `'ab' + 'bc' → 'abbc'`
- ▶ Ripetizione: `'ab' * 3 → 'ababab'`
- ▶ Conversione carattere in codice Unicode:
`ord('a') → 97`
- ▶ Conversione codice Unicode in carattere:
`chr(98) → 'b'`

Apici singoli e doppi

Stringhe delimitate da apici **singoli** possono contenere apici **doppi**

```
>>> print('stringa "protetta" da apici singoli')
stringa "protetta" da apici singoli
```

Stringhe delimitate da apici **doppi** possono contenere apici **singoli**

```
>>> print("stringa che contiene l'apice singolo")
stringa che contiene l'apice singolo
```

Sequenze escape e virgolettatura tripla

Caratteri speciali nelle stringhe possono essere inseriti in due modi:

1. *Sequenze escape*: speciali sequenze di caratteri inizianti per \

\'	apice singolo	\n	a capo
\"	apice doppio	\t	tabulazione (Tab)
\\	carattere '\'		

```
>>> print('\\"'/'\n-*-\n/'\'')
\'/
-*-
/'\
```

2. *Virgolettatura tripla*: delimitando una stringa con **tre** apici, tutti i suoi caratteri, tranne \, sono interpretati alla lettera

```
>>> print('\"_\"_\"')
_\"_\"_
```

Valori booleani (`bool`)

Due soli valori: `True` (1) e `False` (0)

Operazioni logiche: `not`, `and`, `or`

Operazione	Notazione matematica	Sintassi Python
negazione	$\neg x$	<code>not x</code>
congiunzione	$x \wedge y$	<code>x and y</code>
disgiunzione	$x \vee y$	<code>x or y</code>

Controllo di flusso

Operatori relazionali

Relazioni logiche quali

- ▶ == (uguale a)
- ▶ >= (maggiore o uguale a)
- ▶ in (appartiene a)
- ▶ ecc.

forniscono valori **booleani**

```
>>> 1 >= 2
False
>>> 1 == (2 - 1)
True
>>> 'i' in 'ciao'
True
>>> 'e' in 'ciao'
False
```

Operatori relazionali più comuni

- ▶ Uguaglianza: `==` (uguale), `!=` (diverso)
- ▶ Confronto d'ordine: `<`, `>`, `<=`, `>=`
 - ▶ Sui numeri, vale l'ordinamento dei numeri reali
 $-2 < 0.1 < 3 < 1.0e5$
 - ▶ Sulle sequenze, vale l'ordinamento *lessicografico*:
`'a' < 'abc' < 'b' < 'c' < 'caa'`
- ▶ Appartenenza ad una sequenza:
`x in seq`, `x not in seq`

Costrutto condizionale: `if ... else ...`

```
1     if condizione_logica:
2         blocco1          # blocco1 è eseguito se la condizione è vera
3     else:
4         blocco2          # blocco2 è eseguito se la condizione è falsa
5     blocco3              # blocco3 è eseguito in ogni caso
```

L'*indentazione* del codice è fondamentale in Python: **identifica** i *blocchi* di istruzioni da eseguire nei vari casi

Si indenta tramite caratteri Tab (**tabulazione**) o gruppi di spazi

Forme abbreviate di `if ... else ...`

- ▶ Se si omette il blocco `else`, il programma non esegue nessuna speciale istruzione nel caso in cui la condizione sia falsa
- ▶ Se un blocco `if` o `else` consiste di **una sola** riga, per brevità può essere inserito direttamente dopo i due punti

Esempio:

```
1     if a > b:  
2         m = a  
3     else:  
4         m = b
```

può essere abbreviato in:

```
1     if a > b: m = a  
2     else: m = b
```


Condizioni a catena: `elif`

`elif` sta per `else if`; permette di `concatenare` i controlli

```
1  # esame.py
2  voto = int(input('Inserisci il voto: '))
3  if voto < 18:
4      print("mi dispiace")
5  elif voto == 18:
6      print("appena sufficiente")
7  elif voto < 24:
8      print("OK, ma potevi fare meglio")
9  elif voto == 30:
10     print("congratulazioni!")
11 else:
12     print("bene!")
```

Ciclo indefinito: `while`

```
while condizione_logica:  
    blocco
```

1. Valuta una condizione booleana
2. Se vera, esegue un blocco di istruzioni e ricomincia

Esempio: Metodo di Erone

```
1  "Calcolo della radice quadrata con il metodo di Erone"
2
3  def radice(x):
4      "Restituisce y tale che  $x < y^2 \leq x + 1e-12$ "
5      y = x + 1
6      while y**2 - x > 1e-12:
7          y = (y + x/y) / 2
8      return y
9
10 if __name__ == "__main__":
11     x = float(input())
12     r = radice(x)
13     print(r)
```



Erone di Alessandria, 10 D.C. – 70 D.C.

Ciclo definito: `for`

```
for x in range(a, b):  
    blocco
```

esegue il blocco iterando la variabile `x` nell'intervallo che va da `a` (incluso) a `b` (escluso)

Esempio:

```
1     for x in range(1, 11):  
2         print(x)
```

stampa i numeri da 1 a 10

Con un solo argomento: `range(b)`, il ciclo va da `0` (incluso) a `b` (escluso)

Istruzioni `break` e `continue`

Con l'istruzione `break` possiamo forzare l'**uscita** dal ciclo in qualunque momento:

```
1     import math
2     while True:
3         x = float(input())
4         if x < 0:
5             break # esce immediatamente dal ciclo while
6         print(math.sqrt(x))
```

Con l'istruzione `continue` possiamo forzare il **salto** all'iterazione successiva del ciclo:

```
1     for c in range(1,6):
2         if c == 3:
3             continue
4         print(c)
5     # stampa 1, 2, 4, 5
```

Funzioni, moduli e documentazione

Funzioni e moduli

Approccio Python alla programmazione modulare

- ▶ Un *modulo* è un file `.py` contenente un insieme di *funzioni*
- ▶ Una *libreria* è un insieme di moduli
- ▶ La *libreria standard* di Python consiste di tutti i moduli predefiniti

Chiamate a funzione

- ▶ Una funzione ha un *nome* (es.: `abs`)
- ▶ Riceve 0 o più *argomenti in input* (es.: l'intero `-42`)
- ▶ Restituisce 0 o 1 valore di *output* (es.: `42`)
- ▶ Può causare *effetti collaterali (side-effect)* (es.: `print()` non restituisce valori, ma causa una stampa)

Esempi dai moduli `math` e `random`

Funzione	semantica
<code>math.exp(a)</code>	funzione esponenziale (e^a)
<code>math.log(a)</code>	funzione logaritmo ($\log_e a$)
<code>math.pow(a, b)</code>	eleva a alla potenza b (a^b)
<code>math.sqrt(a)</code>	radice quadrata di a (\sqrt{a})
<code>math.e</code>	valore di e (costante)
<code>math.pi</code>	valore di π (costante)
<code>math.inf</code>	valore speciale ∞ (costante)
<code>random.random()</code>	decimale casuale tra 0 ed 1

Utilizzo di un modulo

Prima di poter essere utilizzato, un modulo deve essere *importato* (caricato in memoria) con l'istruzione `import`:

```
import <modulo>
```

```
>>> import math    # carichiamo il modulo delle funzioni matematiche
```

Per invocare una funzione del modulo:

```
nome_modulo.nome_funzione(argomenti)
```

```
>>> math.log10(10)
```

```
1.0
```

```
>>> math.log10(1.0e32)
```

```
32.0
```

```
>>> math.log2(0.5)
```

```
-1.0
```

```
>>> math.sin(math.pi / 2) # math.pi è la costante pi greco
```

```
1.0
```

Importare solo alcune funzioni del modulo

```
from <modulo> import <funz1>, <funz2>, ...
```

importa **solo** le funzioni funz1, funz2 **direttamente** nello spazio globale dei nomi

In questo caso possiamo invocarle con la sintassi abbreviata:

```
nome_funzione(argomenti)
```

```
>>> from math import cos
>>> print(cos(0.4))
0.9210609940028851
```

Documentazione dei moduli

```
>>> import math
>>> help(math)
```

Help on built-in module math:

NAME

math

DESCRIPTION

This module is always available. It provides access to the mathematical functions defined by the C standard.

FUNCTIONS

acos(...)
acos(x)

Return the arc cosine (measured in radians) of x.

acosh(...)
acosh(x)

Return the inverse hyperbolic cosine of x.

<... ALTRE INFORMAZIONI ...>

Documentazione delle funzioni

```
>>> import math
>>> help(math.log)
```

Help on built-in function log in module math:

```
log(...)
    log(x[, base])
```

Return the logarithm of x to the given base.

If the base not specified, returns the natural logarithm (base e) of x.

Definizione di una funzione: `def`

```
def nome_funzione(parametro1, ..., parametroN):  
    corpo
```

- ▶ Intestazione (`def`, nome, parametri)
- ▶ Corpo
 - ▶ Istruzioni `return`

```
1  def cubo(x):      # intestazione: nome e parametri  
2      y = x ** 3  # corpo  
3      return y    # istruzione return
```

Valore di ritorno: `return`

```
return <espressione>
```

Una funzione può *restituire un valore* grazie all'istruzione `return`:

```
1     def doppio_di(x):  
2         return 2 * x
```

In questo caso, l'espressione assume il valore corrispondente:

```
3     a = 7  
4     b = doppio_di(a)  
5     print(b) # stampa '14'
```

Quando viene eseguita la `return`, la funzione invocata **termina immediatamente**

Il flusso di controllo torna all'istruzione *chiamante*

Valore di ritorno

Se la funzione non è terminata da un'istruzione `return`, il suo valore di ritorno è il valore speciale `None`, di tipo `NoneType`

La `return` può restituire solo **un** valore per volta

- ▶ ma può essere un valore di tipo sequenza

Parametri con valori di default

```
def f(a, b, c=espr1, d=espr2): ...
```

- ▶ Un parametro della forma `var = espr` può essere **omesso** durante l'invocazione
- ▶ In tal caso, la funzione è invocata con `var` pari al valore dell'espressione `espr`
- ▶ Se un parametro ha un default, tutti i parametri successivi **devono** averne uno

```
>>> def potenza(b, k=2):  
>>> ... return b ** k  
>>> potenza(10)  
100  
>>> potenza(10, 3)  
1000  
>>> potenza(10, k=4)  
10000
```

Spazio dei nomi

In ogni momento di esecuzione del programma esiste uno *spazio dei nomi*

- ▶ nomi delle variabili e funzioni definite, moduli caricati
- ▶ ad ogni nome corrisponde una sola entità Python

```
>>> temp = 4.2
>>> type(temp)
<class 'float'>
>>> def temp():
...     return 5
>>> type(temp)
<class 'function'>
```

La variabile speciale `__name__`

- ▶ Se un modulo è importato, `__name__` contiene il suo nome
- ▶ Altrimenti, `__name__` è pari a `'__main__'`

```
1  # modulo.py
2  def quadrato(x):
3      return x**2
4
5  def cubo(x):
6      return x**3
7
8  if __name__ == '__main__':
9      print('codice che prova il modulo')
```

Documentazione: le *docstrings*

Docstring: stringa posta all'inizio di un modulo o funzione

```
1 # ARITMETICA.py
2 """
3     Massimo Comun Divisore e funzioni correlate
4 """
5
6 def gcd(a, b):
7     "Restituisce il Massimo Comun Divisore di a e b"
8     ...
9
10 def lcm(a, b):
11     "Restituisce il Minimo Comune Multiplo di a e b"
12     ...
```

La docstring forma la documentazione restituita da `help()`

```
>>> import ARITMETICA
>>> help(ARITMETICA.lcm)
Help on function lcm in module ARITMETICA:

lcm(a, b)
    Restituisce il Minimo Comune Multiplo di a e b
```

Documentazione: le *docstrings*

```
>>> import ARITMETICA
```

```
>>> help(ARITMETICA)
```

```
Help on module ARITMETICA:
```

```
NAME
```

```
ARITMETICA - Massimo Comun Divisore e funzioni correlate
```

```
FUNCTIONS
```

```
gcd(a, b)
```

```
Restituisce il Massimo Comun Divisore di a e b
```

```
lcm(a, b)
```

```
Restituisce il Minimo Comune Multiplo di a e b
```

Tipi sequenza

Tipi sequenza

- ▶ stringhe (`str`)
- ▶ range (`range`)
- ▶ tuple (`tuple`)
- ▶ liste (`list`)
- ▶ ...

Ciascun tipo sequenza permette:

- ▶ accesso tramite indice
- ▶ *slicing*
- ▶ iterazione sugli elementi
- ▶ test di appartenenza

Intervalli interi: `range`

- ▶ `range(b)` rappresenta l'intervallo di interi da 0 incluso a b escluso
- ▶ `range(a,b)` rappresenta l'intervallo di interi da a incluso a b escluso
- ▶ `range(a,b,s)` rappresenta la sequenza di interi da a incluso a b escluso, presi a "passi" di lunghezza s

Es.: `range(2, 12, 3)` rappresenta la sequenza 2, 5, 8, 11

Accesso tramite indice

- ▶ `seq[i]` restituisce l' $(i+1)$ -esimo elemento di `seq`
- ▶ **Primo** elemento: `seq[0]`
- ▶ `seq[-i]` restituisce l' i -esimo elemento della sequenza **rovesciata**

```
>>> s = 'abcdefg'
>>> s[0] # accesso tramite indice
'a'
>>> s[4]
'e'
>>> s[-1] # indicizzazione dal fondo
'g'
>>> range(3,7)[2]
5
```

Slicing: “affettare” una sequenza

- ▶ `seq[i:j]` è la sottosequenza che va dall'indice `i` **incluso**, fino all'indice `j` **escluso**

```
>>> s = 'XYZABC'  
>>> s[1:4]  
'YZA'
```

- ▶ Senza il primo indice, la sottosequenza parte dall'**inizio** della lista

```
>>> s[:4]  
'XYZA'
```

- ▶ Senza il secondo indice, la sottosequenza termina alla **fine** della lista

```
>>> s[1:]  
'YZABC'
```

Slicing di stringhe

Una slice di una stringa è a sua volta una stringa:

```
1     a = 'penna'  
2     b = 'ananas'  
3     c = a + b[3:]  
4     print(c) # stampa 'pennanas'
```

Rovescio di una sequenza

`seq[::-1]` ci dà la sequenza `seq` in ordine **inverso**:

```
>>> 'acetone'[::-1]
'enoteca'
```

Possiamo fare **slicing** direttamente sulla sequenza rovesciata con la sintassi `seq[i:j:-1]` (con $i > j$):

```
>>> 'acetone'[5:1:-1]
'note'
```

Iterazione su una sequenza (for)

```
for variabile in seq:  
    blocco
```

- ▶ La variabile assume di volta in volta un valore della sequenza
- ▶ Il blocco viene eseguito ciascuna volta con quel valore

```
>>> for i in range(0, 5):  
>>> ... print(i ** 2)  
0  
1  
4  
9  
16  
>>> for c in 'pmdr':  
>>> ... print(c + 'o')  
po  
mo  
do  
ro
```

Test di appartenenza

elem in seq

è un test booleano che ritorna **True** se e solo se qualche elemento della sequenza seq è uguale ad elem

```
>>> 7 in range(0, 5):  
False  
>>> 'y' in 'python'  
True
```

Tuple

Una *tupla* è una sequenza di valori indicata con la sintassi (a, b, c, ...), per esempio (10, 25, 30)

Una tupla di un **singolo** elemento è indicata con la sintassi (x,), per esempio (1,)

La *tupla vuota* è indicata con ()

Nelle assegnazioni, le parentesi possono essere omesse:

```
>>> t = (10, 25, 30)
>>> print(t)
(10, 25, 30)
>>> t = 1, 2
>>> print(t)
(1, 2)
```

Assegnazione multipla

Da una sequenza a più variabili

Con un'istruzione della forma

```
target1, target2, ... = seq
```

si realizza un'*assegnazione multipla*:

- ▶ La sequenza `seq` deve avere la **stessa** lunghezza del numero di variabili sulla sinistra
- ▶ Il primo elemento di `seq` viene assegnato a `target1`, il secondo a `target2`, eccetera

Esempio: `a, b, c = 10, 20, 30`

Ciò permette di realizzare anche **scambi**:

```
x, y = y, x # scambia i valori di x e y
```


Liste (list)

Una *lista* è una versione “dinamica” di una tupla

- ▶ Come una tupla, memorizza una sequenza di valori:

```
>>> moschettieri = ['Athos', 'Porthos', 'Aramis']
>>> moschettieri[1]
'Porthos'
```

- ▶ Gli elementi di una lista possono essere **riassegnati**:

```
>>> moschettieri[1] = 'Richelieu'
>>> moschettieri
['Athos', 'Richelieu', 'Aramis']
```

- ▶ È possibile **aggiungere** elementi ad una lista con `.append()`:

```
>>> moschettieri.append("D'Artagnan")
>>> moschettieri
['Athos', 'Richelieu', 'Aramis', "D'Artagnan"]
```

Cancellazione da una lista

- ▶ Per cancellare l'elemento di indice i :
`del lista[i]`
- ▶ Per cancellare l'intera sottolista da i a $j-1$:
`del lista[i:j]`
- ▶ Gli indici degli elementi restanti “slittano” di conseguenza
- ▶ Il metodo `lista.pop()` restituisce l'ultimo elemento e, al contempo, lo cancella dalla lista:

```
>>> moschettieri = ['Athos', 'Porthos', 'Aramis']
>>> x = moschettieri.pop()
>>> moschettieri
['Athos', 'Porthos']
>>> x
'Aramis'
```

Slicing di liste

Lo slicing naturalmente è applicabile alle liste:

```
>>> moschettieri = ['Athos', 'Porthos', 'Aramis', "D'Artagnan"]
>>> moschettieri[1:3]
['Porthos', 'Aramis']
>>> moschettieri[-2:]
['Aramis', "D'Artagnan"]
```

Altre operazioni su liste, tuple e stringhe

- ▶ `len(seq)` restituisce la **lunghezza** di `seq`
- ▶ `seq1 + seq2` restituisce la sequenza ottenuta **concatenando** `seq1` e `seq2`
- ▶ `seq * k` (oppure `k * seq`) restituisce la sequenza ottenuta concatenando ***k* copie** di `seq`

```
>>> len('abcdefghijklmnopqrstuvwxy')
26
>>> 'ca' + 3*'po' + 'lo'
capopopolo
>>> 8 * [0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

Esempio: Rappresentazione di un mazzo di carte

```
1  valore = [ '2', '3', '4', '5', '6', '7', '8', '9', '10', 'J', 'Q', 'K', 'A' ]
2  seme = [ '♥', '♦', '♣', '♠' ]
3  mazzo = 52 * [ ' ' ]
4
5  # Popola il mazzo
6  for j in range(4):
7      for i in range(13):
8          mazzo[i + 13*j] = valore[i] + seme[j]
9
10 # Stampa
11 for carta in mazzo:
12     print(carta)
```

0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
2♥	3♥	4♥	5♥	6♥	7♥	8♥	9♥	10♥	J♥	Q♥	K♥	A♥	2♦	...

Nota: abbiamo usato 4 e 13 per concretezza, ma sarebbe meglio scrivere `len(seme)` e `len(valore)` rispettivamente

Operazioni su lista: costo computazionale

Costi per una lista di n elementi

<https://wiki.python.org/moin/TimeComplexity>

Operazione	Costo nel caso peggiore (ammortizzato)
<code>len(L)</code>	$O(1)$
<code>L[i]</code>	$O(1)$
<code>L[i:i+k]</code>	$O(k)$
<code>L.append(x)</code>	$O(1)$
<code>x = L.pop()</code>	$O(1)$
<code>L.insert(i, x)</code>	$O(n)$
<code>del L[i]</code>	$O(n)$
<code>L.sort()</code>	$O(n \log n)$

Per inserimento in coda e cancellazione in testa, il tipo sequenza `collections.deque` è preferibile a `list`

Entità mutabili e immutabili

- ▶ Un'entità **mutabile** può essere modificata dopo la sua definizione
- ▶ Un'entità **immutabile** no

Sequenze mutabili:

- ▶ liste (**list**)

Sequenze immutabili:

- ▶ stringhe (**str**)
- ▶ tuple (**tuple**)
- ▶ range (**range**)

Entità mutabili e immutabili

```
>>> lista = [1, 2, 3]
```

```
>>> lista[0] = 100
```

```
>>> stringa = 'ciao'
```

```
>>> stringa[0] = 'a'
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'str' object does not support item assignment
```

```
>>> tupla = (1, 2, 3)
```

```
>>> tupla[0] = 100
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'tuple' object does not support item assignment
```

```
>>> R = range(1, 4)
```

```
>>> R[0] = 100
```

```
Traceback (most recent call last):
```

```
File "<stdin>", line 1, in <module>
```

```
TypeError: 'range' object does not support item assignment
```


Assegnazione di entità mutabili: attenzione!

Un'assegnazione di entità mutabile copia **solo** i riferimenti alla zona di memoria contenente il valore assegnato!

Questo può causare gravi effetti indesiderati:

```
>>> A = [10, 20, 30]
>>> B = A # copia implicita -- attenzione!
>>>      # l'entità associata ad A e B è la stessa!
>>> C = list(A) # copia esplicita in una nuova entità lista
>>> A[0] = 50
>>> print(A)
[50, 20, 30]
>>> print(B) # la modifica di A ha avuto effetto anche su B...
[50, 20, 30]
>>> print(C) # ...ma non su C
[10, 20, 30]
```

La *list comprehension* (descrizione di lista)

Possiamo generare una lista con la sintassi

```
[expr for x in seq]
```

- ▶ Il numero di elementi sarà pari a `len(seq)`
- ▶ L'elemento i -esimo sarà dato dall'espressione `expr` valutata con `x = seq[i]`

Esempio:

```
>>> [x ** 2 for x in range(1,6)]  
[1, 4, 9, 16, 25]  
>>> [x + 'o' for x in 'pmdr']  
['po', 'mo', 'do', 'ro']
```

List comprehension con filtro

Possiamo specificare una condizione per selezionare quali elementi concorrono a generare la lista (*filtro*):

```
[expr for x in seq if cond]
```

- ▶ L'elemento i -esimo è incluso nella lista se e solo se `cond` valutata con `x = seq[i]` è `True`

Esempio:

```
>>> [x ** 2 for x in range(1,10) if x % 3 == 0]  
[9, 36, 81]
```

Esempio: Array bidimensionale

Un *array bidimensionale* è una sequenza **doppiamente** indicizzata di valori dello stesso tipo

	0	1	2	3	4	5
0	A	A	C	B	A	C
1	B	B	B	B	A	A
2	C	D	D	B	C	A
3	A	A	A	A	A	A
4	C	C	B	C	B	B
5	A	A	A	B	A	A

Per inizializzare un array bidimensionale di n righe e m colonne:

```
1 A = [[0 for j in range(m)] for i in range(n)]
```

oppure

```
1 A = [m*[0] for i in range(n)]
```

Attenzione: $A = n * [m*[0]]$ non va bene!

$m*[0]$ è mutabile, quindi le sue n “copie” **non sono indipendenti**

Insiemi e dizionari

Insiemi

Una collezione di elementi **distinti** può essere rappresentata tramite il tipo di dati **mutabile set**

- ▶ `{10, 20, 30, 10}` è l'insieme `{10, 20, 30}`
- ▶ `set()` (**non** `{}`) è l'insieme **vuoto**
- ▶ `set(seq)` è l'insieme degli elementi della sequenza `seq`
- ▶ `S.add(elem)` **aggiunge** `elem` all'insieme `S`
- ▶ `S.remove(elem)` **rimuove** `elem` dall'insieme `S`
- ▶ `S | T` restituisce l'**unione** di `S` e `T`
- ▶ `S & T` restituisce l'**intersezione** di `S` e `T`
- ▶ `S - T` restituisce la **differenza** di `S` e `T`

Set comprehension (descrizione di insieme)

Come le liste, anche gli insiemi possono essere costruiti tramite la *comprehension*:

```
{expr for x in seq}
```

```
>>> {x for x in 'abracadabra'}  
{'d', 'a', 'c', 'r', 'b'}  
>>> {x for x in 'abracadabra' if x != 'a'}  
{'d', 'r', 'c', 'b'}
```

Un insieme non è una sequenza

`set` **non** è un tipo sequenza, ma un tipo *collection*

Ciò significa che:

- ▶ `x in set`, `len(set)` e `for x in set` sono operazioni supportate
- ▶ L'indicizzazione e lo slicing **non** sono supportati
- ▶ L'ordine tra gli elementi può **non** essere preservato

```
>>> {'tizio', 'caio', 'sempronio'}  
{'caio', 'sempronio', 'tizio'}
```


Ordinare una collezione

```
sorted(coll)
```

restituisce una nuova lista contenente gli elementi di `coll`

L'ordinamento è quello indotto dall'operatore `<`

```
>>> sorted({20, 10, 40, 30})  
[10, 20, 30, 40]  
>>> sorted(['cab', 'abc', 'c'])  
['abc', 'c', 'cab']
```

Dizionari

Un *dizionario* è una collezione di coppie **chiave: valore**, tale che tutte le chiavi sono **distinte**

```
1 voti = {'alice': 30, 'bobo': 28, 'carlo': 30}
```

Mentre una sequenza è indicizzata da interi, un dizionario è indicizzato da chiavi di qualunque tipo **immutabile**

```
2 print(voti['alice']) # stampa '30'
```

I dizionari sono **mutabili**

Dizionari

- ▶ `dict()`, o anche `{}`, è un dizionario **vuoto**

Se `D` è un dizionario:

- ▶ `D[key] = val` **associa** il valore `val` alla chiave `key`
- ▶ `D[key]` restituisce il valore **associato** alla chiave `key`
- ▶ `del D[key]` **cancella** la chiave `key` e il valore associato

Dizionari

Anche un dizionario è un tipo *collection*:

- ▶ `x in D` verifica se `x` è presente in `D` come **chiave**
- ▶ `len(D)` restituisce il **numero** di coppie chiave-valore esistenti in `D`
- ▶ `for x in D` **itera** sulle chiavi di `D`

Comprehension per i dizionari

Anche i dizionari possono essere costruiti tramite la *comprehension*:

```
{key_expr: value_expr for x in seq}
```

```
>>> d = {n: n**2 for n in range(5)}  
>>> print(d)  
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Input ed output

Astrazioni di input ed output (I/O)

- ▶ Canale di input standard: una sequenza di input illimitata
 - ▶ Default: input del terminale, immesso tramite **tastiera**
 - ▶ Nome canale: `sys.stdin`
 - ▶ ID numerico canale: 0
- ▶ Canale di output standard: una sequenza di output illimitata
 - ▶ Default: output del terminale, emesso tramite **terminale video**
 - ▶ Nome canale: `sys.stdout`
 - ▶ ID numerico canale: 1

Sistema operativo: Riassociare l'I/O su file

Riassociare i canali di ingresso e/o uscita permette di leggere l'ingresso da file e/o di scrivere l'uscita su file

Per riassociare l'ingresso al file `dati-in.txt`:

```
~# python3 hello.py < dati-in.txt
```

Per riassociare l'uscita al file `dati-out.txt`:

```
~# python3 hello.py > dati-out.txt
```

Per riassociare sia l'ingresso che l'uscita:

```
~# python3 hello.py < dati-in.txt > dati-out.txt
```


`input()` e `print()`

- ▶ `s = input()` legge una riga di caratteri dal canale di ingresso, memorizzandola nella variabile `s`
- ▶ `print(s)` scrive il contenuto della variabile `s` sul canale di uscita

Varianti comuni:

- ▶ `s = input('Inserisci il dato: ')` stampa un messaggio e legge una riga memorizzandola in `s`
- ▶ `print(a, b)` scrive il contenuto delle variabili `a` e `b` (separate da uno spazio)

Suddivisione e giunzione di stringhe

Funzioni `str.split()` e `str.join()`

- ▶ Suddivisione: spezza una stringa in più parti:
`str.split('abcde', 'c') → ['ab', 'de']`

```
>>> str.split('Mario Rossi', ' ')
['Mario', 'Rossi']
```

- ▶ Giunzione: compone più parti in una stringa
`str.join('x', ['ab', 'cd']) → 'abxcd'`

```
>>> str.join(' ', ['Rossi', 'Mario'])
'Rossi Mario'
```

- ▶ `str.split(x, y)` può essere abbreviato in `x.split(y)`
- ▶ `str.join(x, y)` può essere abbreviato in `x.join(y)`

Leggere più valori da una riga

Problema: come leggere tre valori dalla **stessa** riga?

Soluzione che utilizza:

- ▶ l'assegnazione multipla
- ▶ il metodo `.split()`

Soluzione:

```
>>> sa, sb, sc = input().split()
>>> a, b, c = int(sa), int(sb), int(sc)
```

Leggere un numero arbitrario di valori da un'unica riga

Problema: come leggere un numero qualunque di valori dalla stessa riga?

Soluzione che utilizza:

- ▶ la list comprehension
- ▶ il metodo `.split()`

Soluzione:

```
>>> L = [int(x) for x in input().split()]
>>> 10 20 -10 -30
>>> print(L)
[10, 20, -10, -30]
```

Iterazione sulle righe dell'input

È spesso utile iterare sulle **righe dell'input**, utilizzando l'oggetto `sys.stdin` del modulo `sys`

- ▶ Ciascun elemento è una **stringa**, corrispondente ad una riga dell'input
- ▶ La stringa **include** il carattere di nuova riga `'\n'` (diversamente dal risultato di `input()`)
- ▶ Per eliminare il carattere `'\n'` si può usare il metodo `.strip()`
- ▶ L'utente può terminare l'input digitando `Ctrl` + `D` (su Linux/Mac) o `Ctrl` + `Z` (su Windows)

Esempio

Finché l'utente immette numeri interi (uno per riga), stamparne il quadrato

```
1 import sys
2 for line in sys.stdin:
3     N = int(line.strip())
4     print(N * N)
```

```
~# python3 quadrati.py
5
25
3
9
4
16
^Z
```

Stampare più valori su una riga

Problema: come stampare più valori sulla **stessa** riga?

Soluzione che utilizza:

- ▶ la list comprehension
- ▶ il metodo `.join()` delle stringhe:
Es.: `' '.join(['a', 'b', 'c'])` restituisce la lista
`'a b c'`

Soluzione:

```
>>> L = [10, 20, -10, -30]
>>> print(' '.join([str(x) for x in L]))
10 20 -10 30
```

Formattazione di stringhe

stringa % sequenza

fornisce una nuova stringa in cui degli speciali **segnaposto** vengono sostituiti con i **valori** presenti nella sequenza:

```
>>> 'Buongiorno %s! Sono le ore %d' % ('Mario', 14)
'Buongiorno Mario! Sono le ore 14'
```

%s	stringa	%d	intero (base 10)
%f	float	%%	carattere '%'

- ▶ Con '%4d', '%3f', ecc. possiamo controllare la **larghezza** del campo dati
- ▶ Con '%.2f', '%.3f', ecc. possiamo controllare il numero di **cifre decimali dopo la virgola**

Opzioni comuni per `print()`

```
print(a, b, c, ..., sep=' ', end='\n', file=sys.stdout)
```

- ▶ `a`, `b`, `c`, ... sono le entità da stampare
- ▶ `sep` è una stringa che separa tra loro le entità (default: spazio)
- ▶ `end` è una stringa appesa in coda (default: nuova riga)
- ▶ `file` è il canale di destinazione (default: `sys.stdout`)

Per esempio, `print('messaggio', file=sys.stderr)` stampa un messaggio sul *canale di errore* (di norma è il terminale, ma è un canale separato rispetto a `sys.stdout`)

Variabili globali, locali e visibilità

Variabili globali

Le variabili del frame primario sono dette *globali*

- ▶ Diversamente da quelle locali, le variabili globali **persistono** tra un'invocazione di funzione ed un'altra
- ▶ Dagli altri frame possono essere lette, ma non riassegnate:

```
>>> N = 137
>>> def f(): print(N)
>>> def g(): N = 12
>>> f()
137
>>> g()
>>> print(N)
137
```

`g()` ha operato su una **diversa** variabile `N` nel **proprio** frame!

Variabili globali

Per riassegnare una variabile globale dall'interno di una funzione, la variabile va **dichiarata** come `global`:

```
>>> def g():
...     global N
...     N = 12
...
>>> g()
>>> print(N)
12
```

Se una variabile globale fa riferimento ad un valore **mutabile**, si può mutarne il contenuto senza dichiararla `global` (ma per **riassegnarla** va dichiarata `global`):

```
>>> L = [10, 20, 30, 40]
>>> def h():
...     L[2] = 1
```

Regole di visibilità (*scope*)

Quando viene eseguita l'istruzione `X = val`:

- ▶ Viene riassegnata la variabile `X` nel frame **locale**, a meno che essa non sia stata dichiarata **global**
- ▶ Se `X` è stata dichiarata **global**, viene riassegnata la variabile `X` nel frame primario (globale)

Regole di visibilità (*scope*)

Quando avviene un riferimento ad X , la variabile viene ricercata:

- ▶ Dapprima nel frame **locale**
- ▶ Poi nei frame delle funzioni che **racchiudono** la funzione locale
- ▶ Infine nel frame **globale**

Se X è stata dichiarata **global**, la ricerca è direttamente nel frame globale

Regole di visibilità (*scope*): Esempio

```
>>> x = 27
>>> def f1():
...     x = 42
...     def f2():
...         print(x)
...     f2()
...
>>> f1()
42
>>> print(x)
27
```

Gestione degli errori

Gestione degli errori

- ▶ Eccezioni
- ▶ Asserzioni

Le eccezioni

Cosa fare se una condizione eccezionale provoca una **situazione non contemplata**?

Esempio: divisione $a//b$ con $b==0$

Le *eccezioni* permettono di gestire simili situazioni senza complicare troppo il codice

Sollevere eccezioni: `raise`

`raise` Eccezione(messaggio)

```
1 def ricerca_inversa(diz, val):
2     for chiave in diz:
3         if diz[chiave] == val:
4             return chiave
5     # se la chiave non c'è...
6     raise LookupError # ...solleva un'eccezione di tipo LookupError
```

L'istruzione `raise` interrompe il normale flusso del programma, sollevando un'eccezione

Tipologie più o meno specifiche: `Exception`, `ValueError`, `ArithmeticError`, `LookupError`, `IndexError`, `MemoryError`, `ZeroDivisionError`, ...

Gestione delle eccezioni: `try ... except ...`

Un'eccezione **interrompe** via via l'esecuzione di ciascuna funzione attiva sullo stack, fin quando :

- ▶ si raggiunge il frame principale (crash), oppure,
- ▶ l'eccezione viene *catturata*, al fine di gestire l'errore

Per catturare le eccezioni si usa la sintassi `try-except`:

```
1 try:
2     b = int(input())
3     x = dividi(a, b)
4 except ZeroDivisionError:
5     print('Non puoi dividere per zero! Continuo con x = a')
6     x = a
7     ...
8 print(x)
```

Assertzioni: `assert`

In fase di `debug` è utile assicurarsi che i dati `soddisfino le condizioni` che ci aspettiamo

Un'asserzione è un'istruzione della forma

```
assert condizione_logica
```

```
1 a = float(input())
2 b = quadrato_di(a)
3 assert b >= 0 # ci aspettiamo che b sia sempre non-negativo
4 c = math.log(b)
```

- ▶ Se la condizione è `False`, viene generata un'eccezione
- ▶ Se la condizione è `True`, la `assert` non ha effetto

Assertzioni

- ▶ Se un'asserzione non è soddisfatta, viene sollevata un'eccezione di tipo `AssertionError`
- ▶ L'indicazione del file e del numero di riga ci permette di capire **quale** asserzione sia stata violata
- ▶ Non ci interessa **catturare** gli `AssertionError`, ma solo scovare la condizione non soddisfatta, per identificare il problema durante il debugging
- ▶ Il controllo delle asserzioni viene di norma **disattivato** una volta che il programma va "in produzione" (`python3 -O`)

Oggetti e classi

Tipi definiti dall'utente: `class`

L'istruzione `class X` definisce un nuovo tipo di dato X:

```
1 class X:
2     a = ... # attributi di classe
3     b = ... #
4
5     def f(self, ...): # metodi di classe
6         ...
7     def g(self, ...):
8         ...
```


Istanze di una classe: `self`

L'invocazione di una classe X crea una nuova *istanza* di tipo X:

```
1 class Cane:
2     zampe = 4 # attributo di classe
3     def verso(self): # metodo
4         print('Bau!')
5
6 fido = Cane() # crea un nuovo oggetto di tipo Cane e lo assegna a fido
```

Il parametro `self` identifica l'*istanza* sulla quale un metodo lavora:

```
7 Cane.verso(fido) # stampa 'Bau!'
8 fido.verso() # equivalente a Cane.verso(fido)
```

NB. `self` non è una parola chiave – qualunque nome per il primo parametro è legale, ma la convenzione è di usare `self`

Attributi di istanza

- ▶ Ogni istanza ha un proprio **spazio dei nomi**
- ▶ In un metodo, un assegnamento a `self` della forma `self.a = val` crea un *attributo di istanza*, di nome `a` e valore `val`

```
1 class Cane:
2     def battezza(self, N):
3         self.nome = N
4
5 fido = Cane()
6 fido.battezza('Fido')
7 print(fido.nome) # stampa 'Fido'
```

Inizializzazione di un'istanza: `__init__`

Per inizializzare gli attributi di un'istanza **quando essa viene creata**, è possibile utilizzare il metodo speciale `__init__`:

```
1 class Cane:
2     def __init__(self, N, A):
3         self.nome = N
4         self.anni = A
5     def verso(self):
6         print(self.nome, 'dice: Bau!')
7
8 fido = Cane('Fido', 2) # invoca Cane.__init__(fido, 'Fido', 2)
9 print(fido.nome) # stampa 'Fido'
10 print(fido.anni) # stampa '2'
11 fido.verso() # stampa 'Fido dice: Bau!'
```

Programmazione ad oggetti

Polimorfismo: un metodo con lo stesso nome può avere un effetto diverso a seconda della classe cui appartiene l'istanza

```
1 class Cane:
2     def verso(self):
3         print('Bau!')
4
5 class Gatto:
6     def verso(self):
7         print('Miao!')
8
9 L = [Cane(), Gatto()]
10 for animale in L:
11     animale.verso() # stampa 'Bau!' e 'Miao!'
```

Overloading (sovraccarico) di operatori

Possiamo personalizzare il comportamento di molti **operatori** definendo dei metodi speciali, ad esempio:

- ▶ `__add__(self, other)` definisce il risultato dell'operazione `self + other`
- ▶ `__mul__(self, other)` definisce il risultato dell'operazione `self * other`
- ▶ `__lt__(self, other)` definisce il risultato del confronto `self < other`
- ▶ `__getitem__(self, item)` definisce il risultato dell'accesso `self[item]`

In effetti, `3 + 2` è equivalente a `int.__add__(3, 2)`

Esempio: Somma di matrici 2×2

```
1 class Mat2x2:
2     def __init__(a, b, c, d):
3         self.a = a
4         self.b = b
5         self.c = c
6         self.d = d
7
8     def __add__(self, other):
9         risultato = Mat2x2(self.a + other.a, self.b + other.b, \
10             self.c + other.c, self.d + other.d)
11         return risultato
```

Se A e B sono di tipo Mat2x2, possiamo ora scrivere ad esempio

$C = A + B$

il che è equivalente a $C = \text{Mat2x2}.__add__(A, B)$

Duck typing

“Se cammina come un’anatra e starnazza come un’anatra, allora è un’anatra”

L'assenza di tipi per le variabili implica che ad una funzione può essere passato **qualsunque** tipo di oggetto, **purché** supporti i metodi invocati su di esso

Per esempio, nella funzione:

```
1 def quadrato_ipotenusa(A, B):  
2     return A*A + B*B
```

A e B possono essere di tipo qualunque, purché per tale tipo siano definite la moltiplicazione (`__mul__`) e l'addizione (`__add__`)

Ereditarietà

- ▶ La sintassi `class X(B)` indica che la classe X è una *sottoclasse* della classe B
- ▶ La sottoclasse X *eredita* tutti metodi dalla classe B
- ▶ La sottoclasse X può avere ulteriori metodi, non presenti in B

```
1 class Cane:
2     def verso(self):
3         print('Bau!')
4
5 class Bassotto(Cane):
6     def colore(self):
7         return 'nero'
8
9 b = Bassotto()
10 b.verso() # stampa 'Bau!'
11 b.colore() # restituisce 'nero'
```


Ereditarietà

- ▶ Una classe può **ridefinire** metodi già presenti nella superclasse
- ▶ Quando un metodo è invocato, viene cercato prima nella classe, poi nella superclasse, poi nella superclasse della superclasse, ecc.

```
1 class Cane:
2     def verso(self):
3         print('Bau!')
4     def ha_la_coda(self):
5         return True
6
7 class Bassotto(Cane):
8     def verso(self):
9         print('Bauuu!')
10
11 b = Bassotto()
12 b.verso() # stampa 'Bauuu!'
13 b.ha_la_coda() # restituisce True
```

Librerie

Installazione ed uso di librerie esterne

Esistono centinaia di librerie (*package*) Python che:

1. Estendono le funzionalità della libreria standard
2. Svolgono compiti specifici (analisi di dati, grafica, calcolo scientifico, comunicazione di rete, elaborazione immagini...)

Non sono parte di Python; vanno installate appositamente

Esempi: Matplotlib, NumPy, SciPy, Pandas, NetworkX, ...

Un indice è disponibile su pypi.org (Python Package Index)

Il comando pip

Il comando pip e/o pip3 è installato con la distribuzione base di Python:

```
vincenzo@euler:~$ pip --version
pip 8.1.1 from /usr/lib/python2.7/dist-packages
  (python 2.7)
vincenzo@euler:~$ pip3 --version
pip 8.1.1 from /usr/lib/python3/dist-packages
  (python 3.5)
```

In questo caso, utilizzerò pip3 per installare package per Python 3

Installazione di un package

Per installare un package, invochiamo:

```
pip3 install --user <libreria>
```

```
vincenzo@euler:~# pip3 install --user matplotlib
Collecting matplotlib
  Downloading https://files.pythonhosted.org/
    packages/81/31/4e261379e0cd4e9bbacfc96b124ebac07
    /matplotlib-2.2.2-cp35-cp35m-manylinux1_x86_64.whl
      (12.6MB)
    100% |*****| 12.6MB
      123kB/s
vincenzo@euler:~#
```

Un package può installarne altri da cui **dipende**
(ad es., MatPlotLib dipende da NumPy)

Elenco dei package installati e rimozione

Per avere l'elenco dei package installati:

```
pip3 list
```

```
vincenzo@euler:~# pip3 list
Package            Version
-----
cyclor             0.10.0
decorator          4.3.0
kiwisolver         1.0.1
matplotlib         3.0.1
mypy               0.641
[...]
```

Per rimuovere un package:

```
pip3 uninstall <libreria>
```

Accesso al namespace dei package

Una volta installato, possiamo importare un package come un normale **modulo**:

```
1 import numpy
2 # Possiamo ora accedere al namespace di numpy scrivendo numpy.xxx
3 A = numpy.array([20, 30, 40])
4
5 # Alternativa:
6 import numpy as np # abbreviazione: np.xxx
7 A = np.array([20, 30, 40])
```

Fine
