# Python Standard Library

## Moduli di uso generale





# Modulo `math` e built-in

*https://docs.python.org/3/library/math.html*

*https://docs.python.org/3/library/stdtypes.html*

*https://docs.python.org/3/library/functions.html*

Buona parte degli operatori e delle funzioni matematiche sono built-in, ossia sono parte del linguaggio.

- Alcune funzioni più specifiche sono tuttavia inserite nel modulo `math`.



# Come leggere i prototipi

**min**(*iterable*, *[, *key*, *default*])

**max**(*arg1, arg2*, **args*, [, *key*])

**round**(*number*[, *ndigits=0*])

- I parametri tra parentesi quadre (*key*, *default*) sono sempre facoltativi.
- **args* rappresenta da 0 ad N parametri facoltativi.
- * si usa anche per indicare che da quel punto in poi tutti i parametri devono essere chiamati *per nome*.
- Questo vale anche quando è usato insieme al nome di un parametri (**args*).
- Inoltre, l'espressione *arg=val*, indica che *val* è il valore di default di *arg* se quest'ultimo non viene specificato.



# min

**min**(*iterable*, *[, *key*, *default*])

**min**(*arg1, arg2*, **args*, [, *key*])

Restituisce il minimo in un iterabile o in una serie di due o più parametri.

```
>>> min(0, 1, 2, 3)
0
>>> min(range(4))
0
```



# max

**max**(*iterable*, *[, *key*, *default*])

**max**(*arg1, arg2*, **args*, [, *key*])

Restituisce il massimo in un iterabile o in una serie di due o più parametri.

```
>>> max(0, 1, 2, 3)
3
>>> max(range(4))
3
```



# E il parametro *key*?

- Specifica una funzione di ordinamento ad un solo parametro.
- Utile se abbiamo a che fare con tipi di dato complessi, o se vogliamo sfruttare un ordinamento particolare.

```
>>> p1 = {'nome': 'Mario', 'eta': 33}
>>> p2 = {'nome': ‘Luigi', 'eta': 44}
>>> max(p1, p2, key=lambda x: x['eta'])
{'nome': ‘Luigi', 'eta': 44}
```



# E il parametro *default*?

› Specifica cosa restituire nel caso l'iterabile non contenga alcun elemento.

```
>>> min([], default=0)
0
```

! Python 2 non supporta il parametro *default*.



# sum

**sum**(*iterable*, [, *start*])

Somma tutti gli elementi di *iterable* e gli aggiunge *start,* se presente.

```
>>> sum(range(4))
6
>>> sum(range(4), 4)
10
```



# Divisione

```
a / b
```

Divide *a* per *b*. Il risultato è *sempre* un float.

```
>>> 3 / 4
0.75
```

! In Python 2 il risultato è intero se *a* e *b* sono interi.

```
>>> 3 / 4
0
```



# Divisione

```
a // b
```

Conta quante volte *b* è contenuto in *a* (divisione intera). A dispetto del nome, non è detto che il risultato sia un intero.

```
>>> 3 // 4
0
>>> 6.4 // 3.1
2.0
```



# Arrotondamento

`math.floor`(*x*)

Restituisce *x* arrotondato per difetto. Il risultato è un intero.

`math.ceil`(*x*)

Restituisce x arrotondato per eccesso. Il risultato è un intero.

**`round`**(*number*[, *ndigits=0*])

Restituisce *number* arrotondato a *ndigits* cifre decimali. Se *ndigits* è 0, il tipo di ritorno è un intero.

! In Python 2 `round()` restituisce sempre un float.



# Arrotondamento

! Python 2 arrotonda in maniera differente da Python 3.

| Python 2 (round half up) | Python 3 (round half even) |
|---|---|
| `>>> round(-1.5)`<br>`-1`<br>`>>> round(-0.5)`<br>`0`<br>`>>> round(0.5)`<br>`1`<br>`>>> round(1.5)`<br>`2` | `>>> round(-1.5)`<br>`-2`<br>`>>> round(-0.5)`<br>`0`<br>`>>> round(0.5)`<br>`0`<br>`>>> round(1.5)`<br>`2` |



# Arrotondamento

- Nel caso l'ultima cifra è pari a 5:
  - Python 2 arrotonda sempre verso l'alto
  - Python 3 arrotonda verso il numero pari più vicino
- Può sembrare strano, ma funziona meglio con la somma di più numeri:

```
>>> a = [-1.5, -0.5, 0.5, 1.5]
>>> sum(a)
0
>>> sum(round(x) for x in a) # Python 2
2.0
>>> sum(round(x) for x in a) # Python 3
0
```



# math.fsum

math.**fsum**(*iterable*)

Restituisce la somma degli elementi di *iterable*. Questa funzione è più accurata della built-in `sum()` quando si lavora con i numeri a virgola mobile.

```
>>> one = [.1] * 10
>>> sum(one)
0.9999999999999999
>>> math.fsum(one)
1.0
```



# Elevamento a potenza

`x ** y`

Restituisce $x^y$ secondo l'implementazione prevista da *x*.

**pow**(*x*, *y*[, *z*])

Equivalente a `x ** y`. Se *z* è presente, restituisce `(x ** y) % z`.

`math.`**pow**(*x*, *y*)

Equivalente a `float(x) ** float(y)`. Diversa da `pow(x, y)` perchè forza il cast a `float`.

`math.`**sqrt**(*x*)

Restituisce la radice quadrata di *x*.



# Ulteriori funzioni (e costanti) matematiche

**`abs`**(*x*)

Restituisce il valore assoluto di *x*.

`math.`**`exp`**(*x*)

Restituisce $e^x$.

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

Restituisce $\log_e(x)$. Se *base* è presente, restituisce $\log_{base}(x)$.

`math.`**`e`**

Restituisce la costante *e*.

`math.`**`sin`**(*x*)

Restituisce il seno di x.

`math.`**`cos`**(*x*)

Restituisce il coseno di x.

`math.`**`tan`**(*x*)

Restituisce la tangente di x.

`math.`**`pi`**

Restituisce la costante π.



# Esercizio: calcolare la distanza dall'origine

- Scrivere uno script in python che calcoli la distanza euclidea dall'origine di un punto $\boldsymbol{x} \in \mathbb{R}^n$ e la stampi a schermo.
- La distanza dall'origine si calcola con la seguente formula:

$$d(\boldsymbol{x}, O) = \sqrt{\sum_{i=1}^{n} x_i^2}$$

- I componenti del vettore devono essere specificati dall'utente tramite *command line arguments*.



# sys.argv

- È una lista degli argomenti che vengono passati da riga di comando.
- `sys.argv[0]` contiene sempre il nome completo del percorso del file eseguito.

**script.py**

```
import sys
print(sys.argv)
```

```
$ python3 script.py 1 two "da 1 a 3"
['script.py', '1', 'two', 'da 1 a 3']
```



# Riferimenti

`x ** y`

Restituisce $x^y$.

$$d(\boldsymbol{x}, O) = \sqrt{\sum_{i=1}^{n} x_i^2}$$

`sys.argv`

Contiene gli argomenti passati da riga di comando.

`math.fsum`(*iterable*)

Restituisce la somma degli elementi di *iterable*.

`math.sqrt`(*x*)

Restituisce la radice quadrata di *x*.



# Soluzione

```
import sys, math
vec = [float(x) for x in sys.argv[1:]]
dist = math.sqrt(math.fsum(x ** 2 \
     for x in vec))
print('Vector: {}'.format(vec))
print('Distance from origin: ' \
     '{:.2f}'.format(dist))
```



# str.format

! In python 3 non è possibile usare % insieme a `print`.

`str.`**`format`**(**args*, ***kwargs*)

Formatta la stringa `str` a seconda dei parametri specificati.

In combinazione con `print` è usata in sostituzione del %:

```
>>> print 'Numero: %d' % 23
'Numero: 23'
>>> print('Numero: {:d}'.format(23))
'Numero: 23'
```



# Modulo `random`

*https://docs.python.org/3/library/random.html*

› Come suggerisce il nome, contiene funzioni utili per generare numeri pseudo-casuali.



# Funzioni di base

`random.seed`([*a*, *version*])

Inizializza il RNG (random number generator) a partire da *a*. Se *a* è omesso oppure è `None`, il seme viene generato automaticamente dal sistema. Il parametro *version* serve per specificare come viene processato *a*.

! In Python 2 non è presente il parametro *version*.

`random.random`()

Restituisce un numero casuale nell'intervallo [0.0, 1.0).



# Esempio

```
>>> import random
>>> random.seed(12)
>>> random.random()
0.4745706786885481
>>> random.random()
0.657472502657253
>>> random.random()
0.6664104711248381
```



# Consigli utili

- Se `random.seed(a)` non viene chiamato, `random.random()` restituisce comunque dei valori casuali.
- Noterete però che i numeri generati sono diversi ad ogni esecuzione del programma.
- Infatti, chiamare `random.seed(a)` garantisce che tutti i numeri generati successivamente saranno identici ad ogni esecuzione.
- Questo è molto utile se avete necessità di lavorare sempre con la stessa sequenza di numeri generati casualmente (es: simulazioni).



# `random.randrange`, `random.randint`

`random.`**`randrange`**(*stop*)

`random.`**`randrange`**(*start*, *stop* [, *step*])

Equivalente a `random.choice(range(start, stop, step))`. Ad esempio, `random.randrange(0, 10, 3)` estrae un numero casuale tra 0, 3, 6 e 9.

`random.`**`randint`**(*a*, *b*)

Equivalente a `random.randrange(a, b+1)`.
Restituisce un numero intero casuale $a \leq N \leq b$.



# `random.sample`, `random.choice`

`random.`**`sample`**(*population*, *k*)

Estrae *k* elementi distinti dall'iterabile *population* se `k > len(population)`. Restituisce una lista.

```
>>> random.sample(range(10), 3)
[2, 9, 4]
```

`random.`**`choice`**(*seq*)

Equivalente a `random.sample(seq, 1)`. Se *seq* è vuota, solleva un `IndexError`.



# random.shuffle

random.**shuffle**(*x*[, *random*])

Riordina casualmente *x*. *random* è una funzione che restituisce valori casuali nell'intervallo [0.0, 1.0), il cui default è random.random.

```
>>> ancora_numeri = [3, 5, 2, 6]
>>> random.shuffle(ancora_numeri)
>>> ancora_numeri
[2, 6, 5, 3]
```



# Esercizio: semifinali

- Scrivere uno script in python che simuli le semifinali di un torneo. Lo script dovrà:
  1. Leggere dai parametri di riga di comando il seed da utilizzare per la generazione dei numeri casuali.
  2. Generare 4 valori casuali *distinti* in [0, 30[. Ogni numero rappresenta un giocatore e la sua abilità.
  3. Formare due coppie.
  4. Stampare le due coppie a schermo.
  5. Per ogni coppia, confrontare i valori: chi ha il valore di abilità più alto è il vincitore di quel confronto.
  6. Stampare i due vincitori a schermo.



# Riferimenti

`sys.argv`

Contiene gli argomenti passati da riga di comando.

`random.seed`([*a*, *version*])

Inizializza il RNG (random number generator).

`random.sample`(*population*, *k*)

Estrae *k* elementi distinti dall'iterabile *population* se `k > len(population)`.



# Soluzione

```
import random as rnd, sys
rnd.seed(int(sys.argv[1]))
players = rnd.sample(range(30), 4)
couples = (players[0:2], players[2:4])
print("Coppie {}, {}“.format( \
    couples[0], couples[1]))
print("Vincitori: {}, {}".format( \
    max(couples[0]), max(couples[1])))
```



# Modulo `collections`

*https://docs.python.org/3/library/collections.html*

- Contiene una serie di classi che definiscono dei contenitori di dati specializzati da affiancare a quelli built-in.



# collections.namedtuple

collections.**namedtuple**(*typename*, *field_names*, *verbose=False*, *rename=False*)

Crea una sottoclasse di `tuple` di nome *typename*. Questa classe consente di accedere ai campi della tupla, oltre che tramite gli indici, anche attraverso gli attributi definiti da *field_names*.

Funzionalmente, diventa l'equivalente di uno `struct` in C.

Attenzione: questa funzione genera una *classe*, non un oggetto!



# Esempio

```
>>> Punto = collections.namedtuple( \
    'Punto', ('ics', 'ipsilon'))
>>> p = Punto(2, ipsilon=3)
>>> p
Punto(ics=2, ipsilon=3)
>>> p[0], p[1]
(2, 3)
>>> p.ics, p.ipsilon
(2, 3)
```



# collections.namedtuple

› *field_names* può essere:

1. Una stringa contenente una serie di nomi separati da spazi e/o virgole:

```
'ics ipsilon'
'ics,ipsilon'
'ics, ipsilon'
```

2. Una sequenza di nomi:

```
['ics', 'ipsilon']
('ics', 'ipsilon')
```



# `collections.namedtuple`

- *verbose*, se impostato a `True`, stampa a schermo la definizione della classe appena creata.
- *rename*, se impostato a `True`, rinomina automaticamente i campi con nomi non validi. Se ad esempio *field_names* è `['abc', 'def', 'ghi', 'abc']`, poiché contiene due nomi non validi (la parola chiave `def`, e `abc` ripetuto due volte), verrà modificato in `['abc', '_1', 'ghi', '_3']`.



# collections.defaultdict

Supponiamo di voler costruire dizionari di liste.

```
>>> heroes = {}
>>> # Magheggi vari
>>> heroes['marvel'].append("Iron Man")
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
KeyError: 'marvel'
```



# collections.defaultdict

Non abbiamo mai creato la chiave `marvel`!

```
>>> heroes = {}
>>> # Magheggi vari
>>> if 'marvel' not in heroes.keys():
...     heroes['marvel'] = list()
>>> heroes['marvel'].append("Iron Man")
>>> heroes['marvel'].append("Thor")
>>> heroes
{'marvel': ['Iron Man', 'Thor']}
```



# collections.defaultdict

collections.defaultdict ci risparmia del lavoro:

```
>>> heroes = collections.defaultdict(list)
>>> heroes['marvel'].append("Iron Man")
>>> heroes['marvel'].append("Thor")
>>> heroes
defaultdict(<class 'list'>, {'marvel': ['Iron
Man', 'Thor']})
```



# collections.defaultdict

`collections.`**`defaultdict`**([*default_factory* [, …]])

Crea un oggetto con l'interfaccia di un dizionario. Utilizza *default_factory* per inizializzare le chiavi mancanti quando vengono accedute per la prima volta. Per le chiavi già esistenti si comporta come un normale dizionario.

Il costruttore è lo stesso di `dict`, l'unica differenza è che il primo parametro posizionale è sempre *default_factory*.

Se *default_factory* è assente o è `None` si comporta come un normale dizionario anche nel caso di chiavi inesistenti (solleva un'eccezione `KeyError`).



# collections.defaultdict

La classe `defaultdict` riscrive il metodo di accesso ai valori in maniera simile a questa:

```
def __getitem__(self, key):
    if key not in self:
        self[key] = self.default_factory()
    return self[key]
```

Quindi non è necessario usare sempre `list()`, ma è in teoria possibile mettere qualsiasi altra funzione senza parametri.



# Modulo `enum`

*https://docs.python.org/3/library/enums.html*

- Contiene la classe `Enum` che definisce enumerazioni, ossia un set di nomi simbolici legati a dei valori costanti e univoci.



# enum.Enum

Per utilizzare `enum.Enum` basta subclassarla.

```
class Color(enum.Enum):
    red = 0
    green = 1
    blue = 2
```

Oppure:

```
class Color(enum.Enum):
    red, green, blue = range(3)
```



# Esempio

```
>>> c = Color.red
>>> c
<Color.red: 0>
>>> c == Color.red
True
>>> c == Color.green
False
```



# Type safety

- Le enumerazione sono type-safe, ossia non sono confrontabili direttamente con altri tipi di dato, a differenza di quanto avviene in linguaggi come il C.

```
>>> type(Color.red)
<enum 'Color'>
>>> Color.red == 1
False
```



# Iterabilità

- Le enumerazioni possono essere iterate per ottenere la sequenza di valori che può assumere.

```
>>> for c in Color:
...  print(c, type(c))
...
Color.red <enum 'Color'>
Color.green <enum 'Color'>
Color.blue <enum 'Color'>
```



# Hashability

- Ogni valore di un'enumerazione può essere convertito in hash (hashable).
- È dunque possibile usare enumerazioni come chiavi nei dizionari.

```
terminal.colors[Color.red] = "#FF0000"
terminal.colors[Color.green] = "#20CC40"
terminal.colors[Color.blue] = "#2020EE"
```



# Python 2

! In Python 2 non esiste il modulo `enum`.

```
>>> import enum
Traceback (most recent call last):
    File "python", line 1, in <module>
ImportError: No module named enum
```



# Python 2

- Vi sono diverse strategie per simulare le enumerazioni.
- La più semplice è quella di ereditare da object:

```
class ColorPy2(object):
    red, green, blue = range(3)
```

- Tuttavia, manca la proprietà di type safety:

```
>>> ColorPy2.red == 0
True
```



# Esercizio: battaglia navale

- In un quadrato 5x5, disporre casualmente 3 navi. Per semplicità, immaginiamo che le navi occupino una sola casella.
- Lo script dovrà chiedere all'utente le coordinate da immettere colpo dopo colpo.
- Determinare se il colpo colpisce il mare (acqua) oppure una delle tre navi (colpito).
- Il gioco termina quando le tre navi sono affondate. Prima di terminare, visualizzare il numero di tentativi effettuati.