



(<http://iplab.dmi.unict.it/>)

Social Media Data Analysis - A.A. 2023-2024

Introduzione a Python per l'analisi dei Dati

Francesco Ragusa - <https://iplab.dmi.unict.it/ragusa/> (<https://iplab.dmi.unict.it/ragusa/>) - <mailto:francesco.ragusa@unict.it>
Antonino Furnari - <http://antoninofurnari.it> (<http://antoninofurnari.it>) - antonino.furnari@unict.it (<mailto:antonino.furnari@unict.it>)

Python è un linguaggio di programmazione ad alto livello, interpretato e pensato per la programmazione "general purpose". Python supporta un sistema dei tipi dinamico e diversi paradigmi di programmazione tra cui la programmazione orientata agli oggetti, la programmazione imperativa, la programmazione funzionale e la programmazione procedurale. Il linguaggio è stato ideato nel 1991 da Guido van Rossum e il suo nome è ispirato alla serie TV satirica Monty Python's Flying Circus (https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus (https://en.wikipedia.org/wiki/Monty_Python%27s_Flying_Circus)).

Benché Python non sia nato come linguaggio di programmazione per il calcolo scientifico e il machine learning, la sua estrema versatilità ha contribuito al nascere di una serie di librerie che rendono la computazione numerica in Python comoda ed efficiente. Buona parte di queste librerie fanno parte di "SciPy" (<https://www.scipy.org/> (<https://www.scipy.org/>)), un ecosistema di software open-source per il calcolo scientifico. Altre librerie, quali ad esempio PyTorch (<http://pytorch.org/> (<http://pytorch.org/>)) mettono a disposizione una serie di strumenti per il calcolo parallelo su GPU orientato al Machine Learning. In queste dispense introdurremo il linguaggio Python 3, vedremo i fondamenti di NumPy, libreria per il calcolo scientifico e Matplotlib, libreria per il plot 2D/3D, e introdurremo gli elementi fondamentali di PyTorch, libreria per il calcolo parallelo su GPU e l'ottimizzazione di algoritmi di machine learning basati sulla discesa del gradiente.

NOTA: Questo laboratorio contiene molte nozioni di base. Il materiale più importante è denotato da un asterisco **.

Referenze importanti da consultare durante il corso, solo le seguenti documentazioni:

- Python 3.9: <https://docs.python.org/3.9/> (<https://docs.python.org/3.9/>);
- Numpy: <http://www.numpy.org/> (<http://www.numpy.org/>);
- Matplotlib: <https://matplotlib.org/> (<https://matplotlib.org/>);
- Pandas: <https://pandas.pydata.org/> (<https://pandas.pydata.org/>).

1 Introduzione a Python

1.1 Numeri

I tipi di dato numerici in Python sono int, float e complex. Noi ci concentreremo su int e float. Alcuni esempi di operazioni tra numeri:

```
In [1]: 1 + 3 #somma
Out[1]: 8

In [2]: 1 - 8 #differenza
Out[2]: -6

In [3]: 1 * 3 #prodotto
Out[3]: 15

In [4]: 1 / 2 #divisione, da notare che in Python 3, La divisione tra numeri interi restituisce un float
Out[4]: 1.5

In [5]: 1 // 2 #divisione intera
Out[5]: 1

In [6]: 1 ** 2 #elevamento a potenza
Out[6]: 81

In [7]: 1 % 2 #modulo
Out[7]: 0

In [8]: (1 + 4) * (3 - 2) #uso delle parentesi
Out[8]: 5
```

1.2 Variabili e Tipi

In generale, i numeri senza virgola vengono interpretati come int, mentre quelli con virgola come float. Dato che Python è tipizzato dinamicamente, non dobbiamo esplicitamente dichiarare il tipo di una variabile. Il tipo verrà associato alla variabile non appena vi assegniamo un valore. Possiamo controllare il tipo di una variabile mediante la funzione type :

```
In [9]: 1 x = 3  
2 y = 9  
3 z = 1.5  
4 h = x/y  
5 l = x//y  
6 type(x), type(y), type(z), type(h), type(l)
```

```
Out[9]: (int, int, float, float, int)
```

E' possibile effettuare il casting da un tipo a un altro mediante le funzioni `int` e `float`:

```
In [10]: 1 int(2.5)
```

```
Out[10]: 2
```

```
In [11]: 1 float(3)
```

```
Out[11]: 3.0
```

Come in C, sono definite le operazioni "in place" tra variabili:

```
In [12]: 1 x = 8  
2 y = 12  
3 x+=y #del tutto equivalente a x=x+y  
4 x
```

```
Out[12]: 20
```

Non sono definite le notazioni `++` e `--`. Per effettuare un incremento di una unità, va utilizzata la notazione `+1`:

```
In [13]: 1 a=1  
2 a+=1  
3 a
```

```
Out[13]: 2
```



Domanda 1

Qual è il tipo della seguente variabile?

```
x = (3//2*2)**2+(3*0.0)
```

Risposta 1



1.3 Booleani

I booleani vengono rappresentati mediante le parole chiave `True` e `False` (iniziano entrambe per maiuscola).

```
In [14]: 1 print(True)  
2 print(False)
```

```
True  
False
```

```
In [15]: 1 type(True)
```

```
Out[15]: bool
```

E' possibile generare booleani mediante gli operatori di confronto:

```
In [16]: 1 5==5
```

```
Out[16]: True
```

```
In [17]: 1 7==5
```

```
Out[17]: False
```

```
In [18]: 1 5>4
```

```
Out[18]: True
```

```
In [19]: 1 9>10
```

```
Out[19]: False
```

```
In [20]: 1 9<=10
```

```
Out[20]: True
```

```
In [21]: 1 11<=10
```

```
Out[21]: False
```

Gli operatori logici sono `and` e `or`:

```
In [22]: 1 print(5==5 and 3<5)
2 print(3>5 or 3<5)
3 print(3>9 or 3<2)
```

```
True
True
False
```

E' possibile effettuare un controllo sui tipi mediante `type` e `==`:

```
In [23]: 1 type(2.5)==float
```

```
Out[23]: True
```

```
In [24]: 1 type(2)==float
```

```
Out[24]: False
```

In alternativa, è possibile utilizzare la funzione `isinstance`:

```
In [25]: 1 isinstance(2.5,float)
```

```
Out[25]: True
```

```
In [26]: 1 isinstance(2.5,int)
```

```
Out[26]: False
```

La funzione `isinstance` è particolarmente comoda quando si vuole controllare che una variabile appartenga a uno tra una serie di tipi. Ad esempio, se vogliamo controllare che una variabile contenga un numero:

```
In [27]: 1 isinstance(2.5,(float,int))
```

```
Out[27]: True
```

```
In [28]: 1 isinstance(5,(float,int))
```

```
Out[28]: True
```

1.4 Stampa

La stampa avviene mediante la funzione `print`:

```
In [29]: 1 var = 2.2
2 print(var)
```

```
2.2
```

Possiamo stampare una riga vuota omettendo il parametro di `print`:

```
In [30]: 1 print(2)
2 print()
3 print(3)
```

```
2
```

```
3
```

Alternativamente, possiamo specificare di inserire due "a capo" alla fine della stampa specificando il parametro `end="\n\n"` (""\n\n" è una stringa - approfondiremo le stringhe in seguito):

```
In [31]: 1 print(2, end="\n\n")
2 print(3)
```

```
2
```

```
3
```

Lo stesso metodo può essere usato per omettere l'inserimento di spazi tra due stampe consecutive:

```
In [32]: 1 print(2, end="") #"" rappresenta una stringa vuota
2 print(3)
```

```
23
```

Possiamo stampare più elementi di seguito separando gli argomenti di `print` con delle virgole. Inoltre, la funzione `print` permette di stampare anche numeri, oltre a stringhe:

```
In [33]: 1 print(1,8/2,7,14%6,True)
```

```
1 4.0 7 2 True
```

1.5 Liste

Le liste sono una struttura dati di tipo sequenziali che possono essere utilizzate per rappresentare sequenze di valori di qualsiasi tipo. Le liste possono anche contenere elementi di tipi misti. Una lista si definisce utilizzando le parentesi quadre:

```
In [34]: 1 l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
2 print(l)
[1, 2, 3, 4, 5]
```

Le liste possono essere indicizzate utilizzando le parentesi quadre. L'indicizzazione inizia da 0 come in C:

```
In [35]: 1 print(l[0],l[2])
2 l[0]=8 #assegnamento di un nuovo valore alla prima locazione di memoria
3 print(l)
1 3
[8, 2, 3, 4, 5]
```

E' possibile aggiungere nuovi valori a una lista mediante la funzione `append`:

```
In [36]: 1 l = []
2 print(l)
3 l.append(1)
4 l.append(2.5)
5 l.append(8)
6 l.append(-12)
7 print(l)
[]
[1, 2.5, 8, -12]
```

Le liste possono essere concatenate mediante l'operatore somma:

```
In [37]: 1 l1 = [1,5]
2 l2 = [4,6]
3 print(l1+l2)
[1, 5, 4, 6]
```

L'operatore di moltiplicazione può essere utilizzato per ripetere una lista. Ad esempio:

```
In [38]: 1 l1 = [1,3]
2 print(l1*2) #concatena l1 a se stessa per due volte
[1, 3, 1, 3]
```

Utilizzando l'operatore di moltiplicazione, è possibile creare velocemente liste con un numero arbitrario di valori uguali. Ad esempio:

```
In [39]: 1 print([0]*5) #Lista di 5 zeri
2 print([0]*4+[1]*1) #4 zeri seguiti da 1 uno
[0, 0, 0, 0, 0]
[0, 0, 0, 0, 1]
```

La lunghezza di una lista può essere ottenuta utilizzando la funzione `len`:

```
In [40]: 1 print(len(12))
2 print(len(12))
[4, 6]
```

L'operatore `==` non controlla se le lunghezze sono uguali, ma verifica che il contenuto delle due liste sia effettivamente uguale:

```
In [42]: 1 print([1,2,3]==[1,2,3])
2 print([1,2,3]==[1,3,2])
True
False
```

E' possibile controllare che un elemento appartenga alla lista mediante la parola chiave `in`:

```
In [43]: 1 print(7 in [1,3,4])
2 print(3 in [1,3,4])
False
True
```

Le funzioni `max` e `min` possono essere utilizzate per calcolare il massimo e il minimo di una lista:

```
In [44]: 1 l=[-5,2,10,6]
2 print(max(l))
3 print(min(l))
10
-5
```

E' possibile rimuovere un valore da una lista mediante il metodo `remove`:

```
In [45]: 1 l=[1,2,3,4,2]
2 print(1)
3 l.remove(2)
4 print(1)

[1, 2, 3, 4, 2]
[1, 3, 4, 2]
```

Tale metodo tuttavia rimuove solo la **prima occorrenza** del valore passato. Se vogliamo rimuovere un valore identificato da uno specifico indice, possiamo usare il costrutto `del`:

```
In [46]: 1 l=[1,2,3,4,2]
2 print(1)
3 del l[4]
4 print(1)

[1, 2, 3, 4, 2]
[1, 2, 3, 4]
```

Inoltre, per accedere all'ultimo elemento e rimuoverlo, possiamo usare il metodo `pop`:

```
In [47]: 1 l=[1,2,3,4,5]
2 print(l)
3 print(l.pop())
4 print(l)

[1, 2, 3, 4, 5]
5
[1, 2, 3, 4]
```

1.5.1 Indicizzazione e Slicing *

E' possibile estrarre una sottolista da una lista specificando il primo indice (incluso) e l'ultimo indice (escluso) separati dal simbolo `:`. Questa notazione è in qualche modo reminiscende del metodo `substr` delle stringhe di C++.

```
In [48]: 1 l = [1,2,3,4,5,6,7,8]
2 print("Lista 1      ->", l)
3 print("l[0:3]      ->", l[0:3]) #dall'indice 0 (incluso) all'indice 3 (escluso)
4 print("l[1:2]      ->", l[1:2]) #dall'indice 1 (incluso) all'indice 2 (escluso)

Lista 1      -> [1, 2, 3, 4, 5, 6, 7, 8]
l[0:3]      -> [1, 2, 3]
l[1:2]      -> [2]
```

Quando il primo indice è omesso, questo viene automaticamente sostituito con "0":

```
In [49]: 1 print("l[:2]      ->", l[:2]) #dall'indice 0 (incluso) all'indice 2 (escluso)
2 #equivale al seguente:
3 print("l[0:2]      ->", l[0:2]) #dall'indice 0 (incluso) all'indice 2 (escluso)

l[:2]      -> [1, 2]
l[0:2]      -> [1, 2]
```

Analogamente, se omettiamo il secondo indice, esso viene sostituito con l'ultimo indice della lista:

```
In [50]: 1 print("Ultimo indice della lista:",len(l))
2 print("l[3:]      ->", l[3:]) #dall'indice 3 (incluso) all'indice 5 (escluso)
3 #equivale al seguente:
4 print("l[3:5]      ->", l[3:8]) #dall'indice 3 (incluso) all'indice 5 (escluso)

Ultimo indice della lista: 8
l[3:]      -> [4, 5, 6, 7, 8]
l[3:5]      -> [4, 5, 6, 7, 8]
```

Omettendo entrambi gli indici:

```
In [51]: 1 print("l[:]      ->", l[:]) #dall'indice 0 (incluso) all'indice 5 (escluso)
2 #equivale a:
3 print("l[0:8]      ->", l[0:8]) #dall'indice 0 (incluso) all'indice 8 (escluso)

l[:]      -> [1, 2, 3, 4, 5, 6, 7, 8]
l[0:8]      -> [1, 2, 3, 4, 5, 6, 7, 8]
```

E' inoltre possibile specificare il "passo", come terzo numero separato da un altro simbolo `:`:

```
In [52]: 1 print("l[0:8:2]      ->", l[0:8:2])
2 #da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento si e uno no)
3 #equivale a:
4 print("l[::2]      ->", l[::2])
5 #da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento si e uno no)

l[0:8:2]      -> [1, 3, 5, 7]
l[::2]      -> [1, 3, 5, 7]
```

Per invertire l'ordine degli elementi, è inoltre possibile specificare un passo negativo. In questo caso, bisogna assicurarsi che il primo indice sia maggiore del secondo:

```
In [53]: 1 print("l[5:2:-1]      ->", l[5:2:-1])
2 #da 5 (incluso) a 2 (escluso) a un passo di -1
3 print("l[2:5:-1]      ->", l[2:5:-1])
4 #in questo caso, il primo indice è più piccolo del secondo,
5 #quindi il risultato sarà una lista vuota

l[5:2:-1]      -> [6, 5, 4]
l[2:5:-1]      -> []
```

Anche in questo caso, omettendo degli indici, questi verranno rimpiazzati con le scelte più ovvie. Nel caso dell'omissione però, cambiano le condizioni di inclusione ed esclusione degli indici. Vediamo qualche esempio:

```
In [54]: 1 print("l[2:-1]      ->", l[2:-1])
2 #dall'ultimo indice (incluso) a 2 (escluso) a un passo di -1
3 #equivale a:
4 print("l[8:2:-1]    ->", l[8:2:-1])
5
6 print()
7 print("l[3::-1]      ->", l[3::-1])
8 #dal terzo indice (incluso) a 0 (incluso, in quanto omesso) a un passo di -1
9 #simile, ma non equivalente a:
10 print("l[3:0:-1]    ->", l[3:0:-1])
11 #dal terzo indice (incluso) a 0 (escluso) a un passo di -1
12
13 print()
14 print("l[::-1]        ->", l[::-1])
15 #dall'ultimo indice (incluso) al primo (incluso, in quanto omesso) a un passo di -1
16 #simile, ma non equivalente a:
17 print("l[8:0:-1]    ->", l[8:0:-1])
18 #dall'ultimo indice (incluso) al primo (escluso) a un passo di -1

l[:2:-1]    -> [8, 7, 6, 5, 4]
l[8:2:-1]   -> [8, 7, 6, 5, 4]

l[3::-1]    -> [4, 3, 2, 1]
l[3:0:-1]   -> [4, 3, 2]

l[::-1]     -> [8, 7, 6, 5, 4, 3, 2, 1]
l[8:0:-1]   -> [8, 7, 6, 5, 4, 3, 2]
```

La notazione `::-1`, in particolare, è utile per invertire le liste:

```
In [55]: 1 print(1)
2 print(l[::-1])

[1, 2, 3, 4, 5, 6, 7, 8]
[8, 7, 6, 5, 4, 3, 2, 1]
```

Indicizzazione e slicing possono essere utilizzate anche per assegnare valori agli elementi delle liste. Ad esempio:

```
In [56]: 1 l = [5,7,9,-1,2,6,5,4,-6]
2 print(l)
3 l[3]=80
4 print(l)

[5, 7, 9, -1, 2, 6, 5, 4, -6]
[5, 7, 9, 80, 2, 6, 5, 4, -6]
```

E' anche possibile assegnare più di un elemento alla volta:

```
In [57]: 1 l[::2]=[0,0,0,0] #assegno 0 ai numeri di posizione dispari
2 print(l)

[0, 7, 0, 80, 0, 6, 0, 4, 0]
```

Le liste possono anche essere annidate:

```
In [58]: 1 a1 = [1,2,[4,8,[7,5]],9],2
2 print(a1)

[1, 2, [4, 8, [7, 5]], [9], 2]
```

L'indicizzazione di queste strutture annidate avviene concatenando gli indici come segue:

```
In [59]: 1 print(a1[2][2][0]) #il primo indice seleziona la lista [4,8,...]
2 #il secondo indice seleziona la lista [7,5]
3 #il terzo indice seleziona l'elemento 7

7
```

Domanda 2



Estrarre la lista `[3, 1.2]` dalla seguente lista:

```
1 = [1, 4, 5, [7, 9, -1, [0, 3, 2, 1.2], 8, []]]
```

Risposta 2



1.6 Tuple

Le tuple sono simili alle liste, ma sono immutabili. Non possono cioè essere modificate dopo la loro inizializzazione. A differenza delle liste, le tuple vengono definite utilizzando le parentesi tonde:

```
In [60]: 1 l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
2 t = (1,2,3,4,5) #questa è una tupla (parentesi tonde)
3 print(l)
4 print(t)
```

```
[1, 2, 3, 4, 5]
(1, 2, 3, 4, 5)
```

Le regole di indicizzazione e slicing viste per le liste valgono anche per tuple. Tuttavia, come accennato prima, le tuple non possono essere modificate:

```
In [61]: 1 t = (1,3,5)
2 try:
3     t[0]=8 #restituisce un errore in quanto le tuple sono immutabili
4 except Exception as e:
5     print(e)
```

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

Inizializzare una tupla con un solo elemento produrrà un numero. Ciò avviene in quanto le parentesi tonde vengono utilizzate anche per raggruppare i diversi termini di una operazione:

```
In [62]: 1 t=(1)
2 print(t)
```

```
1
```

Per definire una tupla monodimensionale, dobbiamo aggiungere esplicitamente una virgola, dopo il primo elemento:

```
In [63]: 1 t=(1,)
2 print(t)
```

```
(1,)
```

E' inoltre possibile omettere le parentesi nella definizione delle tuple:

```
In [64]: 1 t1=1,3,5
2 t2=1,
3 print(t1,t2)
```

```
(1, 3, 5) (1,)
```

E' possibile convertire tuple in liste e viceversa:

```
In [65]: 1 l=[1,2,3,4,5,6,7,8]
2 t=(4,5,6,7,4,8,2,4)
3 ttl = list(t)
4 ltt = tuple(l)
5
6 print(ttl)
7 print(ltt)
```

```
[4, 5, 6, 7, 4, 8, 2, 4]
(1, 2, 3, 4, 5, 6, 7, 8)
```

Le tuple possono inoltre essere create e "spacchettate" al volo:

```
In [66]: 1 t1=1,2,3
2
3 print(t1)
4
5 a,b,c=t1 #spacchettamento della tupla
6 print(a,b,c)
```

```
(1, 2, 3)
1 2 3
```

Questo sistema permette di effettuare lo swap di due variabili in una sola riga di codice:

```
In [67]: 1 var1 = "Var 1"
2 var2 = "Var 2"
3
4 print(var1,var2)
5
6 var1,var2=var2,var1
7 print(var1,var2)
8
9 #equivalente a:
10 var1 = "Var 1"
11 var2 = "Var 2"
12 t = (var2,var1)
13 var1=t[0]
14 var2=t[1]
15 print(var1,var2)
```

```
Var 1 Var 2
Var 2 Var 1
Var 2 Var 1
```

Le tuple annidate possono essere spacciate come segue:

```
In [68]: 
1 t = (1,(2,3),(4,5,6))
2 x,(t11,t12),(t21,t22,t23) = t
3 print(t)
4 print(x, t11, t12, t21, t22, t23)
5 #La notazione a,b,c,d,e,f = t restituirebbe un errore
```

```
(1, (2, 3), (4, 5, 6))
1 2 3 4 5 6
```



Domanda 3

Qual è la differenza principale tra tuple e liste? Si faccia l'esempio di un caso in cui una tupla è un tipo più appropriato rispetto a una lista.

Risposta 3



1.7 Stringhe

In Python possiamo definire le stringhe in tre modi:

```
In [69]: 
1 s1 = 'Singoli apici'
2 s2 = "Doppi apici, possono contenere anche apici singoli '' "
3 s3 = """Tripli
4 doppi apici
5 possono essere definite su più righe"""
6 type(s1), type(s2), type(s3)
```

```
Out[69]: (str, str, str)
```

La stampa avviene mediante la funzione "print":

```
In [70]: 
1 print(s1)
2 print(s2)
3 print(s3)
```

```
Singoli apici
Doppi apici, possono contenere anche apici singoli ''
Tripli
doppi apici
possono essere definite su più righe
```

Le stringhe hanno inoltre una serie di metodi predefiniti:

```
In [71]: 
1 print("ciao".upper()) #rendi tutto maiuscolo
2 print("CIAO".lower()) #tutto minuscolo
3 print("ciao come stai".capitalize()) #prima lettera maiuscola
4 print("ciao come stai".split()) #spezza una stringa e restituisce una lista
5 print("ciao, come stai".split(','))# spezza quando trova la virgola
6 print("-".join(["uno","due","tre"]))#costruisce una stringa concatenando gli elementi
7 #della lista e separandoli mediante il delimitatore
```

```
CIAO
ciao
Ciao come stai
['ciao', 'come', 'stai']
['ciao', ' come stai']
uno-due-tre
```

Le stringhe possono essere indicizzate in maniera simile agli array per ottenere delle sottostringhe:

```
In [72]: 
1 s = "Hello World"
2 print(s[:4]) #primi 4 caratteri
3 print(s[4:]) #dal quarto carattere alla fine
4 print(s[4:7]) #dal quarto al sesto carattere
5 print(s[::-1]) #inversione della stringa
```

```
Hello
o World
o W
dlrow olleH
```

Il metodo `split` in particolare può essere utilizzato per la tokenizzazione o per estrarre sottostringhe in maniera agevole. Ad esempio, supponiamo di voler estrarre il numero 2017 dalla stringa A-2017-B2 :

```
In [73]: 
1 print("A-2017-B2".split('-')[1])
```

```
2017
```

L'operatore `==` controlla che due stringhe siano uguali:

```
In [74]: 1 print("ciao"]=="ciao")
2 print("ciao"]=="ciao2")
```

True
False

Gli altri operatori rispecchiano l'ordinamento lessicografico tra stringhe:

```
In [75]: 1 print("abc"<"def")
2 print("Abc">>"def")
```

True
False



Domanda 4

Quale codice permette di manipolare la stringa azyp-kk9-382 per ottenere la stringa Kk9 ?

Risposta 4



1.7.1 Formattazione di Stringhe *

Possiamo costruire stringhe formattate seguendo una sintassi simile a quella di printf:

```
In [76]: 1 #per costruire la stringa formattata, faccio seguire la stringa dal simbolo "%" e poi inserisco
2 #una tupla contenente gli argomenti
3 s1 = "Questa %s è formattata. Posso inserire numeri, ad esempio %0.2f" % ("stringa",3.00002)
4 print(s1)
```

Questa stringa è formattata. Posso inserire numeri, ad esempio 3.00

Un modo alternativo e più recente di formattare le stringhe consiste nell'utilizzare il metodo "format":

```
In [77]: 1 s2 = "Questa {} è formattata. Posso inserire numeri, ad esempio {}"\n2     .format("stringa",3.00002)# il carattere "\n" permette di spezzare la riga
3 print(s2)
```

Questa stringa è formattata. Posso inserire numeri, ad esempio 3.000002

E' possibile specificare il tipo di ogni argomento utilizzando i due punti:

```
In [78]: 1 print("Questa {:s} è formattata. Posso inserire numeri, ad esempio {:.2f}"\
2     .format("stringa",3.00002)) #parametri posizionali, senza speci
```

Questa stringa è formattata. Posso inserire numeri, ad esempio 3.00

E' anche possibile assegnare nomi agli argomenti in modo da richiamarli in maniera non ordinata:

```
In [79]: 1 print("Questa {str:s} è formattata. Posso inserire numeri, ad esempio {num:.2f}"\
2     .format(num=3.00002, str="stringa"))
```

Questa stringa è formattata. Posso inserire numeri, ad esempio 3.00

Domanda 5

Date le variabili:

```
a = "hello"
b = "world"
c = 2.0
```

Usare la formattazione delle stringhe per stampare la stringa hello 2 times world .

Risposta 5



1.7.2 Formattazione di Stringhe mediante F-Strings *

Le f-string permettono di formattare le stringhe in maniera ancora più compatta ed elegante. In particolare, la notazione "f" permette di includere all'interno di una stringa i valori di variabili contenuti in variabili disponibili nello scope corrente. Esempio:

```
In [80]: 1 x = 2
2 print(f"x={x}")

x=2
```

La 'f' prima della stringa denota l'inizio di una f-string. Il contenuto tra le parentesi graffe viene di fatto valutato come del codice. Ciò permette di fare cose del genere:

```
In [81]: 1 print(f"x={x+2}")

x=4
```

O cose del genere:

```
In [82]: 1 a={'x':2, 'y':'string'}
2 print(f"x={a['x']}, y={a['y']}")

x=2, y=string
```

Anche in questo caso è possibile specificare un formato dopo il simbolo ':' all'interno delle parentesi graffe:

```
In [83]: 1 print(f"x={a['x']:0.2f}")

x=2.00
```

1.7.3 Espressioni Regolari *

Spesso può tornare utile manipolare stringhe mediante le espressioni regolari. Ciò permette di cercare pattern, isolargli ed eliminarli. Le espressioni regolari sono fornite dal pacchetto `re`. Vediamo una stringa di esempio:

```
In [84]: 1 s = "Stringa con numero 029876 e testo"
```

Vediamo ad esempio come estrarre il numero all'interno della stringa:

```
In [85]: 1 import re
2 re.search('[0-9]+',s)
```

```
Out[85]: <re.Match object; span=(18, 24), match='029876'>
```

Il metodo `search` restituisce un oggetto di tipo `re.Match` che specifica gli indici di inizio e fine del match nella stringa (span) e il testo trovato. Per accedere al testo, dobbiamo utilizzare il metodo `group`:

```
In [86]: 1 re.search('[0-9]+',s).group()
```

```
Out[86]: '029876'
```

Il metodo `search` restituisce solo il primo match. In alcuni casi, può essere utile cercare lo stesso pattern più volte all'interno di una stringa. Per farlo, possiamo utilizzare il metodo `finditer`, come nell'esempio seguente:

```
In [87]: 1 s = "Stringa con numeri 029845 18029 numeri formattati 0234-672, 0xxx-1o2 indirizzi@email.com, #hashtags, @names"
2 re.finditer('[0-9]+',s)
```

```
Out[87]: <callable_iterator at 0x7ffbf0132128>
```

Questo restituisce un iterator che permette di accedere ai diversi match. Per estrarre il testo contenuto in ogni match possiamo fare come segue:

```
In [88]: 1 [m.group() for m in re.finditer('[0-9]+',s)]
```

```
Out[88]: ['029845', '18029', '0234', '672', '0', '1', '2']
```

Cerchiamo solo i numeri strutturati del tipo ####-###:

```
In [89]: 1 [m.group() for m in re.finditer('[0-9]{4}-[0-9]{3}',s)]
```

```
Out[89]: ['0234-672']
```

Cerchiamo gli indirizzi email:

```
In [90]: 1 [m.group() for m in re.finditer('[a-zA-Z0-9-_]+@[a-zA-Z0-9-_]+',s)]
```

```
Out[90]: ['indirizzi@email.com']
```

E' possibile rimuovere alcuni pattern mediante il metodo `sub`. Ad esempio, possiamo eliminare le email con:

```
In [91]: 1 re.sub('[a-zA-Z0-9-_]+@[a-zA-Z0-9-_]+',' ',s)
```

```
Out[91]: 'Stringa con numeri 029845 18029 numeri formattati 0234-672, 0xxx-1o2 , #hashtags, @names'
```

Possiamo anche suddividere una riga in più righe sulla base di un pattern. Per esempio, suddividiamo ogni volta che troviamo un trattino:

```
In [92]: 1 re.split('-',s)
```

```
Out[92]: ['Stringa con numeri 029845 18029 numeri formattati 0234',
          '672, 0xxx',
          '1o2 indirizzi@email.com, #hashtags, @names']
```

Suddividiamo ogni volta che troviamo un numero di una o più cifre:

```
In [93]: 1 re.split('[0-9]+',s)
```

```
Out[93]: ['Stringa con numeri ',  
          ' ',  
          ' numeri formattati ',  
          ' ',  
          ' ',  
          'xxx-',  
          'o',  
          ' indirizzi@email.com, #hashtags, @names' ]
```

E' possibile trovare maggiori informazioni nella documentazione ufficiale: <https://docs.python.org/3/library/re.html> (<https://docs.python.org/3/library/re.html>).

1.8 Dizionari *

I dizionari sono simili a delle liste, ma vengono indicizzate da oggetti di tipo "hashable", ad esempio stringhe:

```
In [94]: 1 d = {"val1":1, "val2":2}  
2 print(d)  
3  
4 print(d["val1"])
```

```
{'val1': 1, 'val2': 2}  
1
```

E' possibile ottenere la lista delle chiavi e dei valori come segue:

```
In [95]: 1 print(d.keys()) #chiavi e valori sono in ordine casuale  
2 print(d.values())
```

```
dict_keys(['val1', 'val2'])  
dict_values([1, 2])
```

E' possibile indicizzare dizionari con tuple (che sono "hashable")

```
In [96]: 1 d = {(2,3):5, (4,6):11}  
2  
3 print(d[(2,3)])
```

```
5
```

I dizionari possono anche essere estesi dinamicamente:

```
In [97]: 1 d = dict() #dizionario vuoto  
2 d["chiave"]="valore"  
3 print(d)
```

```
{'chiave': 'valore'}
```

Possiamo controllare che un elemento si trovi tra le chiavi di un dizionario come segue:

```
In [98]: 1 d = {1:'ciao', '5': 5, 8: -1}  
2 print(5 in d)  
3 print('5' in d)
```

```
False  
True
```

E' possibile controllare che un elemento si trovi tra i valori di un dizionario come segue:

```
In [99]: 1 print(-1 in d.values())
```

```
True
```

1.9 Set

I set sono delle strutture dati che possono contenere solo una istanza di un dato elemento:

```
In [100]: 1 s = {1,2,3,3}  
2 print(s) #può essere contenuto solo un "3"
```

```
{1, 2, 3}
```

Possiamo aggiungere un elemento a un set mediante il metodo "add":

```
In [101]: 1 print(s)  
2 s.add(5)  
3 print(s)  
4 s.add(1) #non ha effetto. 1 è già presente  
5 print(s)
```

```
{1, 2, 3}  
{1, 2, 3, 5}  
{1, 2, 3, 5}
```

Anche in questo caso, possiamo controllare l'appartenenza di un elemento ad un set mediante la parola chiave `in`:

```
In [102]: 1 s={1,5,-1}
2 print(-1 in s)
3 print(8 in s)
```

True
False

E' inoltre possibile creare set da liste:

```
In [103]: 1 set([1,3,3,2,5,1])
```

Out[103]: {1, 2, 3, 5}

1.10 Costrutti if/elif/else

I costrutti condizionali funzionano in maniera simile ai linguaggi basati su C. A differenza di tali linguaggi tuttavia, Python sostituisce le parentesi con l'indentazione obbligatoria. Il seguente codice C++:

```
int var1 = 5;
int var2 = 10;
if(var1<var2) {
    int var3 = var1+var2;
    cout << "Hello World " << var3;
}
cout << "End";
```

viene invece scritto come segue:

```
In [104]: 1 var1 = 5
2 var2 = 10
3 if var1<var2:
4     var3 = var1+var2
5     print("Hello World",var3)
6 print("End")
```

Hello World 15
End

In pratica:

- la condizione da verificare non è racchiusa tra parentesi;
- i due punti indicano l'inizio del corpo dell'if;
- l'indentazione stabilisce cosa appartiene al corpo dell'if e cosa non appartiene al corpo dell'if.

Dal momento che l'indentazione ha valore sintattico, essa diventa obbligatoria. Inoltre, non è possibile indentare parti di codice ove ciò non è significativo. Ad esempio, il seguente codice restituirebbe un errore:

```
print("Hello")
    print("World")
```

Le regole di indentazione appena viste, valgono anche per i cicli e altri costrutti in cui è necessario delimitare blocchi di codice. Il costrutto if, permette anche di specificare un ramo else e un ramo elif per i controlli in cascata. Vediamo alcuni esempi:

```
In [105]: 1 true_condition = True
2 false_condition = False
3
4 if true_condition: #i due punti ":" sono obbligatori
5     word="cool!"
6     print(word) #L'indentazione è obbligatoria
7
8 if false_condition:
9     print("not cool :(")
10
11 if not false_condition: #neghiamo la condizione con "not"
12     word="cool"
13     print(word,"again :)")
14
15 if false_condition:
16     word="this"
17     print(word+" is "+false)
18 else: #due punti + indentazione
19     print("true")
20
21 if false_condition:
22     print("false")
23 elif 5>4: #implementa un "else if"
24     print("5>4")
25 else:
26     print("4<5??")
```

cool!
cool again :(
true
5>4

E' anche possibile verificare se un valore appartiene a una lista. Ciò è molto utile per verificare se un parametro è ammesso.

```
In [106]: 1 allowed_parameters = ["single", "double", 5, -2]
2
3 parameter = "double"
4
5 if parameter in allowed_parameters:
6     print(parameter,"is ok")
7 else:
8     print(parameter,"not in",allowed_parameters)
9
10 x=8
11 if x in allowed_parameters:
12     print(x,"is not ok")
13 else:
14     print(x,"not in",allowed_parameters)
```

double is ok
8 not in ['single', 'double', 5, -2]

Una variante "inline" del costrutto if può essere utilizzata per gli assegnamenti condizionali:

```
In [107]: 1 var1 = 5
2 var2 = 3
3 m = var1 if var1>var2 else var2 #calcola il massimo
4 print(m)
```

5

In Python non esiste il costrutto switch, per implementare il quale si utilizza una lista di elif in cascata:

```
In [108]: 1 s = "ciao"
2 if s=="help":
3     print("help")
4 elif s=="world":
5     print("hello",s)
6 elif s=="ciao":
7     print(s,"mondo")
8 else:
9     print("Default")
```

ciao mondo



Domanda 6

Si consideri il seguente codice:

```
x=2
if x>0:
    y=12
    x=x+y
    print y
else:
    z=28
    h=12
    print z+h
```

Si evidenzino eventuali errori sintattici. Si riscriva il codice in C++ o Java e si confrontino le due versioni del codice.

Risposta 6



1.11 Cicli while e for

I cicli while, si definiscono come segue:

```
In [109]: 1 i=0
2 while i<5: #due punti :
3     print(i) #indentazione
4     i+=1
```

0
1
2
3
4

La sintassi dei cicli for è un po' diversa dalla sintassi standard del C. I cicli for in Python sono più simili a dei **foreach** e richiedono un "iterable" (ad esempio una lista o una tupla) per essere eseguiti:

```
In [110]: 1 l=[1,7,2,5]
2 for v in l:
3     print(v)
```

1
7
2
5

Per scrivere qualcosa di equivalente al seguente codice C:

```
for (int i=0; i<5; i++) {...}
```

possiamo utilizzare la funzione `range` che genera numeri sequenziali al volo:

```
In [111]: 1 for i in range(5):
2     print(i)
```

0
1
2
3
4

Range non genera direttamente una lista, ma un "generator" di tipo range, ovvero un oggetto capace di generare numeri. Se vogliamo convertirlo in una lista, dobbiamo farlo esplicitamente:

```
In [112]: 1 print(range(5)) #oggetto di tipo range, genera numeri da 0 a 5 (escluso)
2 print(list(range(5))) #range non fa altro che generare numeri consecutivi
3 #convertendo range in una lista, possiamo verificare quali numeri vengono generati
```

range(0, 5)
[0, 1, 2, 3, 4]

Se volessimo scorrere contemporaneamente indici e valori di un array potremmo scrivere:

```
In [113]: 1 array=[1,7,2,4,5]
2 for i in range(len(array)):
3     print(i,"->",array[i])
```

0 -> 1
1 -> 7
2 -> 2
3 -> 4
4 -> 5

In Python però, è possibile utilizzare la funzione `enumerate` per ottenere lo stesso risultato in maniera più compatta:

```
In [114]: 1 for index,value in enumerate(array): #get both index and value
2     print(index,"->",value)
```

0 -> 1
1 -> 7
2 -> 2
3 -> 4
4 -> 5

Supponiamo adesso di voler scorrere contemporaneamente tutti gli i-esimi elementi di più liste. Ad esempio:

```
In [115]: 1 a1=[1,6,2,5]
2 a2=[1,8,2,7]
3 a3=[9,2,5,2]
4
5 for i in range(len(a1)):
6     print(a1[i],a2[i],a3[i])
```

1 1 9
6 8 2
2 2 5
5 7 2

Una funzione molto utile quando si lavora con i cicli è `zip`, che permette di raggruppare gli elementi corrispondenti di diverse liste:

```
In [116]: 1 l1 = [1,6,5,2]
2 l2 = [3,8,9,2]
3 zipped=list(zip(l1,l2))
4 print(zipped)
```

[(1, 3), (6, 8), (5, 9), (2, 2)]

In pratica, `zip` raggruppa gli elementi i-esimi delle liste in tuple. La i-esima tupla di `zipped` contiene gli i-esimi elementi delle due liste. Combinando `zip` con un ciclo for, possiamo ottenere il seguente risultato:

```
In [117]: 1 for v1,v2,v3 in zip(a1,a2,a3):
2     print(v1,v2,v3)
```

1 1 9
6 8 2
2 2 5
5 7 2

che è equivalente al codice visto in precedenza. E' anche possibile combinare `zip` e `enumerate` come segue:

```
In [118]: 1 for i,(v1,v2,v3) in enumerate(zip(a1,a2,a3)):
2     print(i,"->",v1,v2,v3)
```

0 -> 1 1 9
1 -> 6 8 2
2 -> 2 2 5
3 -> 5 7 2

Attenzione, anche `zip`, come `range`, produce un generator. Pertanto è necessario convertirlo esplicitamente in una lista per stamparne i valori:

```
In [119]: 1 print(zip(l1,l2))
2 print(list(zip(l1,l2)))
```

<zip object at 0x7ffbf014f108>
[(1, 3), (6, 8), (5, 9), (2, 2)]

1.12 Comprensione di liste e dizionari *

La comprensione di liste è uno strumento sintattico che permette di definire liste al volo a partire da altre liste, in maniera iterativa. Ad esempio, è possibile moltiplicare tutti gli elementi di una lista per un valore, come segue:

```
In [120]: 1 a = list(range(8))
2
3 b = [x*3 for x in a] #La lista "a" viene iterata. La variabile "x" conterrà di volta in volta i valori di a
4
5 print(a)
6 print(b)

[0, 1, 2, 3, 4, 5, 6, 7]
[0, 3, 6, 9, 12, 15, 18, 21]
```

E' anche possibile includere solo alcuni elementi selettivamente:

```
In [121]: 1 print([x for x in a if x%2==0]) #include solo i numeri pari

[0, 2, 4, 6]
```

```
In [122]: 1 a = [1,3,8,2,9]
2 b = [4,9,2,1,4]
3
4 c = [x+y for x,y in zip(a,b)]
5
6 print(a)
7 print(b)
8 print(c) #i suoi elementi sono le somme degli elementi di a e b

[1, 3, 8, 2, 9]
[4, 9, 2, 1, 4]
[5, 12, 10, 3, 13]
```

I meccanismi di comprensione si possono utilizzare anche nel caso dei dizionari:

```
In [123]: 1 a = ["one", "two", "three", "four", "five", "six", "seven"]
2 b = range(1,8)
3 d = {i:s for i,s in zip(a,b)}
4 print(d)

{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5, 'six': 6, 'seven': 7}
```



Domanda 7

Tutte le operazioni che si possono fare mediante comprensione di liste e dizionari possono essere fatte mediante un ciclo for? Quali sono i vantaggi principali di queste tecniche rispetto a l'utilizzo dei cicli for?

Risposta 7



1.13 Definizione di Funzioni

Considerato quanto già detto sulla indentazione, la definizione di una funzione è molto naturale:

```
In [124]: 1 def fun(x, y):
2     return x**y
3
4 print(fun(3,2)) #il valore di default "2" viene utilizzato

9
```

Inoltre, in maniera simile a quanto avviene con il linguaggio C, è possibile definire valori di default per i parametri:

```
In [125]: 1 def fun(x, y=2):
2     return x**y
3
4 print(fun(3)) #il valore di default "2" viene utilizzato

9
```

I parametri di una funzione possono essere specificati in un ordine diverso rispetto a quello in cui essi sono stati definiti richiamandone il nome:

```
In [126]: 1 print(fun(y=3,x=2))

8
```

E' possibile definire una funzione che restituisce più di un elemento utilizzando le tuple:

```
In [127]: 1 def soMuchFun(x,y):
2     return x**y, y**x
3
4 print(soMuchFun(2,3))
5
6 a,b=soMuchFun(2,3) #posso "spacchettare" la tupla restituita
7 print(a,b)
```

(8, 9)
8 9

E' inoltre possibile definire funzioni anonime come segue:

```
In [128]: 1 myfun = lambda x: x**2 #un input e un output
2 print(myfun(2))
3
4 myfun1 = lambda x,y: x+y #due input e un output
5 print(myfun1(2,3))
6
7 myfun2 = lambda x,y: (x**2,y**2) #due input e due output
8 print(myfun2(2,3))
```

4
5
(4, 9)

Domanda 8



Qual è il vantaggio di definire una funzione mediante **lambda**? Quali sono i suoi limiti?

Risposta 8



1.14 Map e Filter

Le funzioni **map** e **filter** permettono di eseguire operazioni sugli elementi di una lista. In particolare, **map** applica una funzione a tutti gli elementi di una lista:

```
In [129]: 1 def pow2(x):
2     return(x**2)
3
4 l1 = list(range(6))
5 l2 = list(map(pow2,l1)) #applica pow2 a tutti gli elementi della lista
6 print(l1)
7 print(l2)
```

[0, 1, 2, 3, 4, 5]
[0, 1, 4, 9, 16, 25]

Quando si utilizzano **map** e **filter**, tornano particolarmente utili le funzioni anonime, che ci permettono di scrivere in maniera più compatta. Ad esempio, possiamo riscrivere quanto visto sopra come segue:

```
In [130]: 1 l2 = list(map(lambda x: x**2, l1))
2 print(l2)
```

[0, 1, 4, 9, 16, 25]

Filter permette di selezionare un sottoinsieme degli elementi di una lista sulla base di una condizione. La condizione viene specificata passando una funzione che prende in input l'elemento della lista e restituisce un booleano:

```
In [131]: 1 print(list(filter(lambda x: x%2==0,l1))) #filtra solo i numeri pari
```

[0, 2, 4]

1.15 Programmazione Orientata agli Oggetti - Definizione di Classi

La programmazione orientata agli oggetti in Python è intuitiva. Le principali differenze rispetto ai più diffusi linguaggi di programmazione orientata agli oggetti sono le seguenti:

- non esistono i modificatori di visibilità (**private**, **protected** e **public**). Per convenzione, tutti i simboli privati vanno preceduti da `__`;
- ogni metodo è una funzione con un argomento di default (**self**) che rappresenta lo stato dell'oggetto;
- il costruttore si chiama `__init__`.

```
In [132]: 1 class Classe(object): #ereditiamo dalla classe standard "object"
2     def __init__(self, x): #costruttore
3         self.x=x #inserisco il valore x nello stato dell'oggetto
4
5     def prt(self):
6         print(self.x)
7
8     def power(self,y=2):
9         self.x = self.x**y
10
11 c = Classe(3)
12 c.prt()
13 c.power(3)
14 c.prt()
15 c.power()
16 c.prt()

3
27
729
```

Per estendere una classe, si fa come segue:

```
In [133]: 1 class Classe2(Classe): #derivo la classe "Classe" ed eredito metodi e proprietà
2     def __init__(self,x):
3         super(Classe2, self).__init__(x) #chiamo il costruttore della classe madre
4     def prt(self): #ridefinisco il metodo prt
5         print("yeah",self.x)
6
7 c2 = Classe2(2)
8 c2.power()
9 c2.prt()

yeah 4
```

1.16 Duck Typing

Per identificare i tipi dei dati, Python segue il principio del [duck typing](https://en.wikipedia.org/wiki/Duck_typing) (https://en.wikipedia.org/wiki/Duck_typing). Secondo tale principio, i tipi sono definiti utilizzando il `_duck test_`:

If it looks like a duck, swims like a duck, and quacks like a duck, then it probably is a duck.

Ciò significa che i tipi sono definiti in relazione alle operazioni che possono essere eseguite su di essi. Vediamo un esempio (preso da Wikipedia).

```
In [134]: 1 class Sparrow(object):
2     def fly(self):
3         print("Sparrow flying")
4
5 class Airplane(object):
6     def fly(self):
7         print("Airplane flying")
8
9 class Whale(object):
10    def swim(self):
11        print("Whale swimming")
12
13 def lift_off(entity):
14     entity.fly()
15
16 sparrow = Sparrow()
17 airplane = Airplane()
18 whale = Whale()
19
20 try:
21     lift_off(sparrow)
22     lift_off(airplane)
23     lift_off(whale) #Errore, il "tipo" di questo oggetto non permette di eseguire il metodo "fly".
24             #Secondo il duck test, questo tipo è incompatibile
25 except AttributeError as e:
26     print("Error:",e)
27

Sparrow flying
Airplane flying
Error: 'Whale' object has no attribute 'fly'
```

1.17 Eccezioni *

In maniera simile a molti linguaggi moderni, Python supporta l'uso delle eccezioni. E' possibile catturare una eccezione con il costrutto `try - except`:

```
In [135]: 1 try:
2     5/0
3 except:
4     print("Houston, abbiamo un problema!")

Houston, abbiamo un problema!
```

In Python, le eccezioni sono tipizzate. Possiamo decidere quali tipi di eccezioni catturare come segue:

```
In [136]: 1 try:
2     5/0
3 except ZeroDivisionError:
4     print("Houston, abbiamo un problema!")

Houston, abbiamo un problema!
```

Possiamo avere accesso all'eccezione scatenata (ad esempio ottenere maggiori informazioni) come segue:

```
In [137]: 1 try:  
2     5/0  
3 except ZeroDivisionError as e:  
4     print("Houston, abbiamo un problema!")  
5     print(e)
```

```
Houston, abbiamo un problema!  
division by zero
```

E' possibile lanciare una eccezione mediante `raise`:

```
In [138]: 1 def div(x,y):  
2     if y==0:  
3         raise ZeroDivisionError("Cannot divide by zero!")  
4     else:  
5         return x/y  
6 try:  
7     div(5,2)  
8     div(5,0)  
9 except Exception as e:  
10    print(e)
```

```
Cannot divide by zero!
```

Possiamo definire nuove eccezioni estendendo la classe `Exception`:

```
In [139]: 1 class MyException(Exception):  
2     def __init__(self, message):  
3         self.message = message  
4  
5     try:  
6         raise MyException("Exception!")  
7 except Exception as e:  
8     print(e)
```

```
Exception!
```

Un modo veloce e comodo per lanciare una eccezione (un `AssertionError` nello specifico) quando qualcosa va male, è utilizzare `assert`, che prende in input un booleano e, opzionalmente, un messaggio di errore. Il booleano va posto uguale a `False` se qualcosa è andato storto. Vediamo un esempio:

```
In [140]: 1 def div(x,y):  
2     assert y!=0, "Cannot divide by zero!"  
3     return x/y  
4  
5 try:  
6     div(5,2)  
7     div(5,0)  
8 except:  
9     print('Assert failed')
```

```
Assert failed
```

1.18 Definizione di Moduli

Quando si costruiscono programmi complessi, può essere utile raggruppare le definizioni di funzione e classi in moduli. Il modo più semplice di definire un modulo in Python consiste nell'inserire le definizioni all'interno di un file apposito `modulo.py`. Le definizioni potranno poi essere importati mediante la sintassi `from modulo import funzione`, a patto che il file che richiama le funzioni e quello che definisce il modulo si trovino nella stessa cartella. Vediamo un esempio:

```
#file modulo.py  
def mysum(a,b):  
    return a+b  
  
def myprod(a,b):  
    return a*b  
  
#file main.py (stessa cartella di modulo.py)  
from modulo import mysum, myprod  
print(mysum(2,3)) #5  
print(myprod(2,3)) #6
```

Altre informazioni su usi più avanzati di moduli e pacchetti possono essere reperite qui: <https://docs.python.org/3/tutorial/modules.html> (<https://docs.python.org/3/tutorial/modules.html>).

2 Numpy

Numpy è la libreria di riferimento di **scipy** per il calcolo scientifico. Il cuore della libreria è costituito dai **numpy array** che permettono di gestire agevolmente operazioni tra vettori e matrici. Gli array di Python sono in generale **tensori**, ovvero strutture numeriche dal numero di dimensioni variabili, che possono dunque essere array monodimensionali, matrici bidimensionali, o strutture a più dimensioni (es. cuboidi 10 x 10 x 10). Per utilizzare gli array di numpy, dobbiamo prima importare il pacchetto **numpy**:

```
In [141]: 1 import numpy as np #La notazione "as" ci permette di referenziare il namespace numpy semplicemente con np in futuro
```

2.1 Numpy Arrays *

Un array multidimensionale di numpy può essere definito a partire da una lista di liste, come segue:

```
In [142]: 1 l = [[1,2,3],[4,5,2],[1,8,3]] #una Lista contenente tre Liste
2 print("List of lists:",l) #viene visualizzata così come l'abbiamo definita
3 a = np.array(l) #costruisco un array di numpy a partire dalla lista di liste
4 print("Numpy array:\n",a) #ogni lista interna viene identificata come una riga di una matrice bidimensionale
5 print("Numpy array from tuple:\n",np.array((1,2,3),(4,5,6))) #posso creare numpy array anche da tuple
```

```
List of lists: [[1, 2, 3], [4, 5, 2], [1, 8, 3]]
Numpy array:
[[1 2 3]
 [4 5 2]
 [1 8 3]]
Numpy array from tuple:
[[1 2 3]
 [4 5 6]]
```

Ogni array di numpy ha una proprietà `shape` che ci permette di determinare il numero di dimensioni della struttura:

```
In [143]: 1 print(a.shape) #si tratta di una matrice 3 x 3
```

```
(3, 3)
```

Vediamo qualche altro esempio

```
In [144]: 1 array = np.array([1,2,3,4])
2 matrice = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
3 tensore = np.array([[[1,2,3,4],[['a','b','c','d']],[[5,4,2,3],[['a','b','c','d']],[[7,5,3,2],[['a','b','c','d']],[[0,2,3,1],[['a','b','c','d']]]]])
```

```
print('Array:',array, array.shape) #array monodimensionale, avrà una sola dimensione
print('Matrix:\n',matrice, matrice.shape)
print('matrix:\n',tensore, tensore.shape) #tensore, avrà due dimensioni
```

Vediamo qualche operazione tra numpy array:

```
In [145]: 1 a1 = np.array([1,2,3,4])
2 a2 = np.array([4,3,8,1])
3 print("Sum:",a1+a2) #somma tra vettori
4 print("Elementwise multiplication:",a1*a2) #moltiplicazione tra elementi corrispondenti
5 print("Power of two:",a1**2) #quadrato degli elementi
6 print("Elementwise power:",a1**a2) #elevamento a potenza elemento per elemento
7 print("Vector product:",a1.dot(a2)) #prodotto vettoriale
8 print("Minimum:",a1.min()) #minimo dell'array
9 print("Maximum:",a1.max()) #massimo dell'array
10 print("Sum:",a2.sum()) #somma di tutti i valori dell'array
11 print("Product:",a2.prod()) #prodotto di tutti i valori dell'array
12 print("Mean:",a1.mean()) #media di tutti i valori dell'array
```

```
Sum: [ 5  5 11  5]
Elementwise multiplication: [ 4  6 24  4]
Power of two: [ 1  4  9 16]
Elementwise power: [    1      8 6561     4]
Vector product: 38
Minimum: 1
Maximum: 4
Sum: 16
Product: 96
Mean: 2.5
```

Operazioni tra matrici:

```
In [146]: 1 m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
2 m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])
3
4 print("Sum:",m1+m2) #somma tra matrici
5 print("Elementwise product:\n",m1*m2) #prodotto elemento per elemento
6 print("Power of two:\n",m1**2) #quadrato degli elementi
7 print("Elementwise power:\n",m1**m2) #elevamento a potenza elemento per elemento
8 print("Matrix multiplication:\n",m1.dot(m2)) #prodotto matriciale
9 print("Minimum:",m1.min()) #minimo
10 print("Maximum:",m1.max()) #massimo
11 print("Minimum along columns:",m1.min(0)) #minimo per colonne
12 print("Minimum along rows:",m1.min(1)) #minimo per righe
13 print("Sum:",m1.sum()) #somma dei valori
14 print("Mean:",m1.mean()) #valore medio
15 print("Diagonal:",m1.diagonal()) #diagonale principale della matrice
16 print("Transposed:\n",m1.T) #matrice trasposta
```

Sum: [[9 4 4 8]
[5 8 8 4]
[11 9 5 2]
[0 3 11 7]]
Elementwise product:
[[8 4 3 16]
[0 16 12 3]
[28 20 6 0]
[0 2 24 6]]
Power of two:
[[1 4 9 16]
[25 16 4 9]
[49 25 9 4]
[0 4 9 1]]
Elementwise power:
[[1 4 3 256]
[1 256 64 3]
[2401 625 9 1]
[1 2 6561 1]]
Matrix multiplication:
[[20 26 51 30]
[48 37 57 42]
[68 48 59 45]
[12 21 26 8]]
Minimum: 0
Maximum: 7
Minimum along columns: [0 2 2 1]
Minimum along rows: [1 2 2 0]
Sum: 47
Mean: 2.9375
Diagonal: [1 4 3 1]
Transposed:
[[1 5 7 0]
[2 4 5 2]
[3 2 3 3]
[4 3 2 1]]

2.2 Linspace, Arange, Zeros, Ones, Eye e Random *

Le funzioni **linspace**, **arange**, **zeros**, **ones**, **eye** e **random** di **numpy** sono utili a generare array numerici al volo. In particolare, la funzione **linspace**, permette di generare una sequenza di **n** numeri equispaziati che vanno da un valore minimo a un valore massimo:

```
In [147]: 1 a=np.linspace(10,20,5) # genera 5 valori equispaziati che vanno da 10 a 20
2 print(a)
```

[10. 12.5 15. 17.5 20.]

arange è molto simile a **range**, ma restituisce direttamente un array di **numpy**:

```
In [148]: 1 print(np.arange(10)) #numeri da 0 a 9
2 print(np.arange(1,6)) #numeri da 1 a 5
3 print(np.arange(0,7,2)) #numeri pari da 0 a 6
```

[0 1 2 3 4 5 6 7 8 9]
[1 2 3 4 5]
[0 2 4 6]

Possiamo creare array contenenti zero o uno di forme arbitrarie mediante **zeros** e **ones**:

```
In [149]: 1 print(np.zeros((3,4)))#zeros e ones prendono come parametro una tupla contenente le dimensioni desiderate
2 print(np.ones((2,1)))
```

[[0. 0. 0. 0.]
[0. 0. 0. 0.]
[0. 0. 0. 0.]]
[[1.]
[1.]]

La funzione **eye** permette di creare una matrice quadrata identità:

```
In [150]: 1 print(np.eye(3))
2 print(np.eye(5))
```

[[1. 0. 0.]
[0. 1. 0.]
[0. 0. 1.]]
[[1. 0. 0. 0. 0.]
[0. 1. 0. 0. 0.]
[0. 0. 1. 0. 0.]
[0. 0. 0. 1. 0.]
[0. 0. 0. 0. 1.]]

Per costruire un array di valori casuali (distribuzione uniforme) tra 0 e 1 (zero incluso, uno escluso), basta scrivere:

```
In [151]: 1 print(np.random.rand(5)) #un array con 5 valori casuali tra 0 e 1  
2 print(np.random.rand(3,2)) #una matrice 3x2 di valori casuali tra 0 e 1
```

```
[0.2702533 0.52855705 0.82430719 0.99905097 0.76738557]  
[[0.84779247 0.2424875 ]  
[0.46516776 0.28358575]  
[0.28519302 0.29411536]]
```

Possiamo generare un array di valori casuali distribuiti in maniera normale (Gaussiana) con **randn**:

```
In [152]: 1 print(np.random.randn(5,2))
```

```
[[ 0.62930799 -0.89210824]  
[-0.15464818 -0.13463005]  
[-0.52705111 -1.37919243]  
[ 1.85208699 -0.78306297]  
[ 2.41591078  0.90399183]]
```

Possiamo generare numeri interi compresi tra un minimo (incluso) e un massimo (escluso) usando **randint**:

```
In [153]: 1 print(np.random.randint(0,50,3))#tre valori compresi tra 0 e 50 (escluso)  
2 print(np.random.randint(0,50,(2,3)))#matrice 2x3 di valori interi casuali tra 0 e 50 (escluso)
```

```
[30 49 44]  
[[ 4 28 24]  
[14 35 28]]
```

Per generare valori casuali in maniera replicabile, è possibile specificare un seed:

```
In [154]: 1 np.random.seed(123)  
2 print(np.random.rand(5))
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897]
```

Il codice sopra (inclusa la definizione del seed) restituisce gli stessi risultati se rieseguito:

```
In [155]: 1 np.random.seed(123)  
2 print(np.random.rand(5))
```

```
[0.69646919 0.28613933 0.22685145 0.55131477 0.71946897]
```

2.3 max, min, sum, argmax, argmin

È possibile calcolare massimi e minimi e somme di una matrice per righe e colonne come segue:

```
In [156]: 1 mat = np.array([[1,-5,0],[4,3,6],[5,8,-7],[10,6,-12]])  
2 print(mat)  
3 print()  
4 print( mat.max(0))# massimi per colonne  
5 print()  
6 print(mat.max(1))# massimi per righe  
7 print()  
8 print( mat.min(0))# minimi per colonne  
9 print()  
10 print(mat.min(1))# minimi per righe  
11 print()  
12 print(mat.sum(0))# somma per colonne  
13 print()  
14 print(mat.sum(1))# somma per righe  
15 print()  
16 print(mat.max())# massimo globale  
17 print(mat.min())# minimo globale  
18 print(mat.sum())# somma globale
```

```
[[ 1 -5  0]  
[ 4  3  6]  
[ 5  8 -7]  
[10  6 -12]]
```

```
[10 8 6]  
[ 1 6 8 10]  
[ 1 -5 -12]  
[ -5 3 -7 -12]  
[ 20 12 -13]  
[ -4 13 6 4]
```

```
10  
-12  
19
```

È inoltre possibile ottenere gli indici in corrispondenza dei quali si hanno massimi e minimi usando la funzione **argmax**:

```
In [157]: 1 print(mat.argmax(0))
```

```
[3 2 1]
```

Abbiamo un massimo alla quarta riga nella prima colonna, uno alla terza riga nella seconda colonna e uno alla seconda riga nella terza colonna. Verifichiamolo:

```
In [158]: 1 print(mat[3,0])
2 print(mat[2,1])
3 print(mat[1,2])
4 print(mat.max(0))

10
8
6
[10  8  6]
```

Analogamente:

```
In [159]: 1 print(mat.argmax(1))
2 print(mat.argmin(0))
3 print(mat.argmin(1))

[0 2 1 0]
[0 0 3]
[1 1 2 2]
```

2.4 Indicizzazione e Slicing *

Gli array di numpy possono essere indicizzati in maniera simile a quanto avviene per le liste di Python:

```
In [160]: 1 arr = np.array([1,2,3,4,5])
2 print("arr[0]      ->",arr[0]) #primo elemento dell'array
3 print("arr[:3]     ->",arr[:3]) #primi tre elementi
4 print("arr[1:5:2]  ->",arr[1:4:2]) #dal secondo al quarto (incluso) a passo 2

arr[0]      -> 1
arr[:3]     -> [1 2 3]
arr[1:5:2]  -> [2 4]
```

Quando si indica un array a più di una dimensione con un unico indice, viene in automatico indicizzata la prima dimensione. Vediamo qualche esempio con le matrici bidimensionali:

```
In [161]: 1 mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
2 print("Matrice:\n",mat,mat.shape) #matrice 3 x 4
3 print("mat[0]      ->",mat[0]) #una matrice è una collezione di righe, per cui mat[0] restituisce la prima riga
4 print("mat[-1]    ->",mat[-1]) #ultima riga
5 print("mat[::-2]  ->",mat[::-2]) #righe dispari

Matrice:
[[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]] (3, 4)
mat[0]      -> [1 5 2 7]
mat[-1]    -> [1 5 2 1]
mat[::-2]  -> [[1 5 2 7]
 [1 5 2 1]]
```

Vediamo qualche esempio con tensori a più dimensioni:

```
In [162]: 1 tens = np.array(([1,5,2,7],
2                  [2,7,3,2],
3                  [1,5,2,1]),
4                  ([[1,5,2,7],
5                  [2,7,3,2],
6                  [1,5,2,1]]))
7 print("Matrice:\n",tens,tens.shape) #tensore 2x3x4
8 print("tens[0]    ->",tens[0])#si tratta della prima matrice 3x4
9 print("tens[-1]   ->",tens[-1])#ultima mtrice 3x4

Matrice:
[[[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]]

 [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]] (2, 3, 4)
tens[0]    -> [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]]
tens[-1]   -> [[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]]
```

L'indicizzazione può proseguire attraverso le altre dimensioni specificando un ulteriore indice in parentesi quadre o separando i vari indici con la virgola:

```
In [163]: mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
1 print("Matrice:\n",mat,mat.shape) #matrice 3 x 4
2 print("mat[2][1] ->",mat[2][1]) #terza riga, seconda colonna
3 print("mat[0,0] ->",mat[0,0]) #prima riga, prima colonna (notazione più compatta)
4
5
6 print("mat[0] -> ",mat[0]) #restituisce l'intera prima riga della matrice
7 print("mat[:,0] -> ",mat[:,0]) #restituisce la prima colonna della matrice.
8
9 print("mat[0,:] ->",mat[0,:]) #I due punti ":" significano "lascia tutto inalterato lungo questa dimensione"
10 print("mat[0:2,:] ->\n",mat[0:2,:]) #prime due righe
11 print("mat[:,0:2] ->\n",mat[:,0:2]) #prime due colonne
12 print("mat[-1] ->",mat[-1]) #ultima riga
```

```
Matrice:
[[1 5 2 7]
 [2 7 3 2]
 [1 5 2 1]] (3, 4)
mat[2][1] -> 5
mat[0,0] -> 1
mat[0] -> [1 5 2 7]
mat[:,0] -> [1 2 1]
mat[0,:] -> [1 5 2 7]
mat[0:2,:] ->
 [[1 5 2 7]
 [2 7 3 2]]
mat[:,0:2] ->
 [[1 5]
 [2 7]
 [1 5]]
mat[-1] -> [1 5 2 1]
```

Caso di tensori a più dimensioni:

```
In [164]: mat=np.array([[[1,2,3,4],[['a','b','c','d']],
1   [[5,4,2,3],[['a','b','c','d']]],
2   [[7,5,3,2],[['a','b','c','d']]],
3   [[0,2,3,1],[['a','b','c','d']]])
4
5 print("mat[:, :, 0] ->", mat[:, :, 0]) #matrice 3 x 3 contenuta nel "primo canale" del tensore
6 print("mat[:, :, 1] ->", mat[:, :, 1]) #matrice 3 x 3 contenuta nel "secondo canale" del tensore
7 print("mat[..., 0] ->", mat[..., 0]) #matrice 3 x 3 contenuta nel "primo canale" del tensore (notazione alternativa)
8 print("mat[..., 1] ->", mat[..., 1]) #matrice 3 x 3 contenuta nel "secondo canale" del tensore (notazione alternativa)
9 #La notazione "... serve a dire "lascia tutto invariato lungo le dimensioni omesse"
```

```
mat[:, :, 0] -> [[[1' 'a']
 ['5' 'a']
 ['7' 'a']
 ['0' 'a']]]
mat[:, :, 1] -> [[[2' 'b']
 ['4' 'b']
 ['5' 'b']
 ['2' 'b']]]
mat[..., 0] -> [[[1' 'a']
 ['5' 'a']
 ['7' 'a']
 ['0' 'a']]]
mat[..., 1] -> [[[2' 'b']
 ['4' 'b']
 ['5' 'b']
 ['2' 'b']]
```

In genere, quando da un array si estraе un sottoinsieme di dati, si parla di **slicing**.

2.4.1 Indicizzazione e Slicing Logici

In numpy è inoltre possibile indicizzare gli array in maniera "logica", ovvero passando come indici un array di valori booleani. Ad esempio, se vogliamo selezionare il primo e il terzo valore di un array, dobbiamo passare come indici l'array [True, False, True] :

```
In [165]: x = np.array([1,2,3])
1 print(x[np.array([True,False,True])]) #per selezionare solo 1 e 3
2 print(x[np.array([False,True,False])]) #per selezionare solo 2
```

```
[1 3]
[2]
```

L'indicizzazione logica è molto utile se combinata alla possibilità di costruire array logici "al volo" specificando una condizione che gli elementi di un array possono o non possono soddisfare. Ad esempio:

```
In [166]: x = np.arange(10)
1 print(x)
2 print(x>2) #genera un array di valori booleani
3 #che conterrà True in presenza dei valori di x
4 #che verificano la condizione x>2
5 print(x==3) #True solo in presenza del valore 3
```

```
[0 1 2 3 4 5 6 7 8 9]
[False False False True True True True True True]
[False False False True False False False False]
```

Unendo questi due principi, è semplice selezionare solo alcuni valori da un array, sulla base di una condizione:

```
In [167]: x = np.arange(10)
1 print(x[x%2==0]) #seleziona i valori pari
2 print(x[x%2!=0]) #seleziona i valori dispari
3 print(x[x>2]) #seleziona i valori maggiori di 2
```

```
[0 2 4 6 8]
[1 3 5 7 9]
[3 4 5 6 7 8 9]
```

2.5 Reshape *

In alcuni casi può essere utile cambiare la "shape" di una matrice. Ad esempio, una matrice 3x2 può essere modificata riarrangiando gli elementi in modo da ottenere una matrice 2x3, una matrice 1x6 o una matrice 6x1. Ciò si può fare mediante il metodo "reshape":

```
In [168]: 1 mat = np.array([[1,2],[3,4],[5,6]])
2 print(mat)
3 print(mat.reshape(2,3))
4 print(mat.reshape(1,6))
5 print(mat.reshape(6,1)) #matrice 6 x 1
6 print(mat.reshape(6)) #vettore unidimensionale
7 print(mat.ravel())#equivalente al precedente, ma aparametrico

[[1 2]
 [3 4]
 [5 6]]
[[1 2 3]
 [4 5 6]]
[[1 2 3 4 5 6]]
[[1]
 [2]
 [3]
 [4]
 [5]
 [6]]
[1 2 3 4 5 6]
[1 2 3 4 5 6]
```

Notiamo che, se leggiamo per righe (da sinistra verso destra, dall'alto verso il basso), l'ordine degli elementi resta immutato. Possiamo anche lasciare che numpy ci accolga una delle dimensioni sostituendola con -1:

```
In [169]: 1 print(mat.reshape(2,-1))
2 print(mat.reshape(-1,6))

[[1 2 3]
 [4 5 6]]
[[1 2 3 4 5 6]]
```

Reshape può prendere in input le singole dimensioni o una tupla contenente la shape. Nell'ultimo caso, risulta comodo fare operazioni di questo genere:

```
In [170]: 1 mat1 = np.random.rand(3,2)
2 mat2 = np.random.rand(2,3)
3 print(mat2.reshape(mat1.shape)) #diamo a mat2 la stessa shape di mat1

[[0.72904971 0.43857224]
 [0.0596779  0.39804426]
 [0.73799541 0.18249173]]
```

2.6 Composizione di Array Mediante concatenate e stack

Numpy permette di unire diversi array mediante due principali funzioni: concatenate e vstack . La funzione concatenate prende in input una lista (o tupla) di array e permette di concatenarli lungo una dimensione (axis) esistente specificata, che di default è pari a zero (concatenazione per righe):

```
In [171]: 1 a=np.arange(9).reshape(3,3)
2 print(a,a.shape,"\n")
3 cat=np.concatenate([a,a])
4 print(cat,cat.shape,"\n")
5 cat2=np.concatenate([a,a,a])
6 print(cat2,cat2.shape)

[[0 1 2]
 [3 4 5]
 [6 7 8]] (3, 3)

[[0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]] (6, 3)

[[0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]
 [0 1 2]
 [3 4 5]
 [6 7 8]] (9, 3)
```

E' possibile concatenare array su una dimensione diversa specificandola mediante il parametro axis :

```
In [172]: 1 a=np.arange(9).reshape(3,3)
2 print(a,a.shape,"\\n")
3 cat=np.concatenate([a,a], axis=1) #concatenazione per colonne
4 print(cat,cat.shape,"\\n")
5 cat2=np.concatenate([a,a,a], axis=1) #concatenazione per colonne
6 print(cat2,cat2.shape)
```

```
[[0 1 2]
[3 4 5]
[6 7 8]] (3, 3)

[[0 1 2 0 1 2]
[3 4 5 3 4 5]
[6 7 8 6 7 8]] (3, 6)

[[0 1 2 0 1 2 0 1 2]
[3 4 5 3 4 5 3 4 5]
[6 7 8 6 7 8 6 7 8]] (3, 9)
```

Affinché la concatenazione sia compatibile, gli array della lista devono avere lo stesso numero di dimensioni lungo quelle che **non vengono concatenate**:

```
In [173]: 1 print(cat.shape,a.shape) #concatenazione Lungo L'asse 0, Le dimensioni Lungo gli altri assi devono essere uguali
2
3 (3, 6) (3, 3)
```

La funzione `stack`, a differenza di `concatenate` permette di concatenare array lungo una nuova dimensione. Si confrontino gli output delle due funzioni:

```
In [174]: 1 cat=np.concatenate([a,a])
2 print(cat,cat.shape)
3 stack=np.stack([a,a])
4 print(stack,stack.shape)
```

```
[[0 1 2]
[3 4 5]
[6 7 8]
[0 1 2]
[3 4 5]
[6 7 8]] (6, 3)

[[[0 1 2]
[3 4 5]
[6 7 8]]]

[[[0 1 2]
[3 4 5]
[6 7 8]]] (2, 3, 3)
```

Nel caso di `stack`, gli array sono stati concatenati lungo una nuova dimensione. E' possibile specificare dimensioni alternative come nel caso di `concatenate`:

```
In [175]: 1 stack=np.stack([a,a],axis=1)
2 print(stack,stack.shape)
3
4 [[[0 1 2]
[0 1 2]]]
5
6 [[[3 4 5]
[3 4 5]]]
7
8 [[[6 7 8]
[6 7 8]]]] (3, 2, 3)
```

In questo caso, gli array sono stati concatenati lungo la seconda dimensione.

```
In [176]: 1 stack=np.stack([a,a],axis=2)
2 print(stack,stack.shape)
3
4 [[[0 0]
[1 1]
[2 2]]]
5
6 [[[3 3]
[4 4]
[5 5]]]
7
8 [[[6 6]
[7 7]
[8 8]]]] (3, 3, 2)
```

In questo caso, gli array sono stati concatenati lungo l'ultima dimensione.

2.7 Tipi

Ogni array di numpy ha il suo tipo (si veda <https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html> (<https://docs.scipy.org/doc/numpy-1.13.0/user/basics.types.html>) per la lista di tipi supportati). Possiamo vedere il tipo di un array ispezionando la proprietà `dtype`:

```
In [177]: 1 print(mat1.dtype)
2
3 float64
```

Possiamo specificare il tipo in fase di costruzione dell'array:

```
In [178]: 1 mat = np.array([[1,2,3],[4,5,6]],int)
2 print(mat.dtype)
3
4 int64
```

Possiamo inoltre cambiare il tipo di un array al volo utilizzando `astype`. Questo è comodo ad esempio se vogliamo effettuare una divisione non intera:

```
In [179]: 1 print(mat/2)
2 print(mat.astype(float)/2)

[[0.5 1. 1.5]
 [2. 2.5 3. ]]
[[0.5 1. 1.5]
 [2. 2.5 3. ]]
```

2.8 Gestione della Memoria in Numpy

Numpy gestisce la memoria in maniera dinamica per questioni di efficienza. Pertanto, un assegnamento o una operazione di slicing, in genere **non creano una nuova copia dei dati**. Si consideri ad esempio questo codice:

```
In [180]: 1 a=np.array([[1,2,3],[4,5,6]])
2 print(a)
3 b=a[0,0:2]
4 print(b)

[[1 2 3]
 [4 5 6]]
[1 2]
```

L'operazione di slicing `b=a[0,0:2]` ha solo permesso di ottenere una nuova "vista" di una parte di `a`, ma i dati non sono stati replicati in memoria. Pertanto, se modifichiamo un elemento di `b`, la modifica verrà applicata in realtà ad `a`:

```
In [181]: 1 b[0]=-1
2 print(b)
3 print(a)

[-1 2]
[[-1 2 3]
 [ 4 5 6]]
```

Per evitare questo genere di comportamenti, è possibile utilizzare il metodo `copy` che forza numpy a creare una nuova copia dei dati:

```
In [182]: 1 a=np.array([[1,2,3],[4,5,6]])
2 print(a)
3 b=a[0,0:2].copy()
4
5 print(b)
6 b[0]=-1
7 print(b)
8 print(a)

[[1 2 3]
 [4 5 6]]
[1 2]
[-1 2]
[[1 2 3]
 [4 5 6]]
```

In questa nuova versione del codice, `a` non viene più modificato alla modifica di `b`.

2.9 Broadcasting

Numpy gestisce in maniera intelligente le operazioni tra array che presentano shape diverse sotto determinate condizioni. Vediamo un esempio pratico: supponiamo di avere una matrice 2×3 e un array 1×3 :

```
In [183]: 1 mat=np.array([[1,2,3],[4,5,6]],dtype=np.float)
2 arr=np.array([2,3,8])
3 print(mat)
4 print(arr)

[[1. 2. 3.]
 [4. 5. 6.]]
[2 3 8]
```

Supponiamo adesso di voler dividere, elemento per elemento, tutti i valori di ogni riga della matrice per i valori dell'array. Possiamo eseguire l'operazione richiesta mediante un ciclo for:

```
In [184]: 1 mat2=mat.copy() #copia il contenuto della matrice per non sovrascriverla
2 for i in range(mat2.shape[0]):#individua le righe
3     mat2[i]=mat2[i]/arr
4 print(mat2)

[[0.5      0.66666667 0.375    ]
 [2.       1.66666667 0.75     ]]
```

```
In [185]: 1 arr.shape
```

Out[185]: (3,)

Se non volessimo utilizzare cicli for, potremmo replicare `arr` in modo da ottenere una matrice 2×3 e poi effettuare una semplice divisione elemento per elemento:

```
In [186]: 1 arr2=np.stack([arr,arr])
2 print(arr2)
3
4 print(mat/arr2)

[[2 3 8]
 [2 3 8]]
[[0.5      0.66666667 0.375    ]
 [2.       1.66666667 0.75     ]]
```

Lo stesso risultato si può ottenere semplicemente chiedendo a numpy di dividere `mat` per `arr`:

In [187]:

```
1 print(mat/arr)
[[0.5      0.66666667 0.375
 [2.       1.66666667 0.75      ]]
```

Ciò avviene in quanto **numpy** confronta le dimensioni dei due operandi (2×3 e 1×3) e adatta l'operando con shape più piccola a quello con shape più grande, replicandone gli elementi lungo la dimensione unitaria (la prima). Il broadcasting in pratica generalizza le operazioni tra scalari e vettori/matrici del tipo:

In [188]:

```
1 print(2*mat)
2 print(2*arr)
[[ 2.  4.  6.]
 [ 8. 10. 12.]]
[ 4  6 16]
```

In generale, quando vengono effettuate operazioni tra due array, numpy compara le shape dimensione per dimensione, dall'ultima alla prima. Due dimensioni sono compatibili se:

- Sono uguali;
- Una di loro è uguale a uno.

Inoltre, le due shape non devono avere necessariamente lo stesso numero di dimensioni.

Ad esempio, le seguenti shape sono compatibili:

$$\begin{array}{c} 2 \times 3 \times 5 \\ 2 \times 3 \times 5 \\ 2 \times 3 \times 5 \\ 2 \times 1 \times 5 \\ 2 \times 3 \times 5 \\ 3 \times 5 \\ 2 \times 3 \times 5 \\ 3 \times 1 \end{array}$$

Vediamo altri esempi di broadcasting:

In [189]:

```
1 mat1=np.array([[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]])
2 mat2=np.array([[2,1,3],[7,6,2]])
3 print("Mat1 shape",mat1.shape)
4 print("Mat2 shape",mat2.shape)
5 print()
6 print("Mat1\n",mat1)
7 print()
8 print("Mat2\n",mat2)
9 print()
10 print("Mat1*Mat2\n",mat1*mat2)
```

```
Mat1 shape (2, 2, 3)
Mat2 shape (2, 3)
```

```
Mat1
[[[1 3 5]
 [7 6 2]]]
```

```
[[6 5 2]
 [8 9 9]]]
```

```
Mat2
[[2 1 3]
 [7 6 2]]]
```

```
Mat1*Mat2
[[[ 2  3 15]
 [49 36  4]]]
```

```
[[12  5  6]
 [56 54 18]]]
```

Il prodotto tra i due tensori è stato effettuato moltiplicando le matrici bidimensionali `mat1[0,...]` e `mat2[0,...]` per `mat2`. Ciò è equivalente a ripetere gli elementi di `mat2` lungo la dimensione mancante ed effettuare un prodotto punto a punto tra `mat1` e la versione adattata di `mat2`.

```
In [190]: 1 mat1=np.array([[[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]]])
2 mat2=np.array([[1,3,5],[6,5,2]])
3 print("Mat1 shape",mat1.shape)
4 print("Mat2 shape",mat2.shape)
5 print()
6 print("Mat1\n",mat1)
7 print()
8 print("Mat2\n",mat2)
9 print()
10 print("Mat1*Mat2\n",mat1*mat2)
```

Mat1 shape (2, 2, 3)
 Mat2 shape (2, 1, 3)

Mat1
 [[[1 3 5]
 [7 6 2]]]

[[[6 5 2]
 [8 9 9]]]

Mat2
 [[[1 3 5]]]
 [[6 5 2]]]

Mat1*Mat2
 [[[1 9 25]
 [7 18 10]]]
 [[36 25 4]
 [48 45 18]]]

In questo caso, il prodotto tra i due tensori è stato ottenuto moltiplicando tutte le righe delle matrici bidimensionali `mat1[0,...]` per `mat2[0]` (prima riga di mat2) e tutte le righe delle matrici bidimensionali `mat1[1,...]` per `mat2[1]` (seconda riga di mat2). Ciò è equivalente a ripetere tutti gli elementi di `mat2` lungo la seconda dimensione (quella contenente 1) ed effettuare un prodotto punto a punto tra `mat1` e la versione adattata di `mat2`.

3. Matplotlib

Matplotlib è la libreria di riferimento per la creazione di grafici in Python. Si tratta di una libreria molto potente e ben documentata (<https://matplotlib.org/>). Vediamo alcuni esempi classici di utilizzo della libreria.

3.1 Plot bidimensionale *

Vediamo come stampare la funzione:

$$y = x^2.$$

Per stampare la funzione, dovremo fornire a matplotlib una serie di coppie di valori (x, y) che verifichino l'equazione riportata sopra. Il modo più semplice per farlo consiste nel:

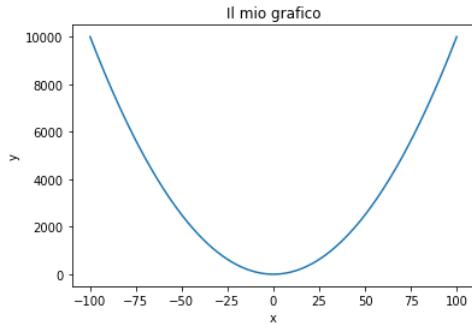
- Definire un vettore arbitrario di valori x estratti dal dominio della funzione;
- Calcolare i rispettivi punti y utilizzando la forma analitica riportata sopra;
- Eseguire il plot dei valori mediante matplotlib.

I due vettori possono essere definiti così:

```
In [323]: 1 x = np.linspace(-100,100,300) #campioniamo in maniera lineare 300 punti tra -100 e 100
2 y = x**2 #calcoliamo il quadrato di ognuno dei punti di x
```

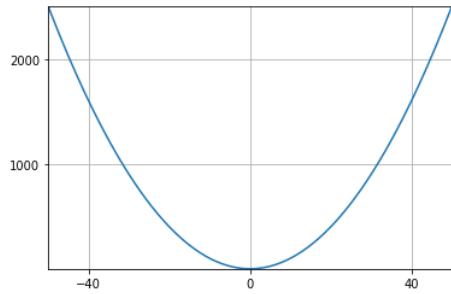
Procediamo alla stampa:

```
In [324]: 1 from matplotlib import pyplot as plt #importo il modulo pyplot da matplotlib e lo chiamo plt
2
3 plt.plot(x,y) #stampa le coppie x,y come punti nello spazio cartesiano
4 plt.xlabel('x') #imposta una etichetta per l'asse x
5 plt.ylabel('y') #imposta una etichetta per l'asse y
6 plt.title('Il mio grafico') #imposta il titolo del grafico
7 plt.show() #mostra il grafico
```



E' possibile controllare diversi aspetti del plot, tra cui i limiti degli assi x e y, la posizione dei "tick" (le lineette che orizzontali e verticali sugli assi) o aggiungere una griglia. Vediamo qualche esempio:

```
In [193]: 1 plt.plot(x,y) #stampiamo lo stesso plot
2 plt.xlim([-50,50]) #visualizziamo i valori x solo tra -20 e 40
3 plt.ylim([0,2500]) #e i valori y solo tra 0 e 4000
4 plt.xticks([-40,0,40]) #inseriamo solo -40, 0 e 40 come "tick" sull'asse x
5 plt.yticks([1000,2000]) #solo 1000 e 2000 come tick y
6 plt.grid() #aggiungiamo una griglia
7 plt.show()
```



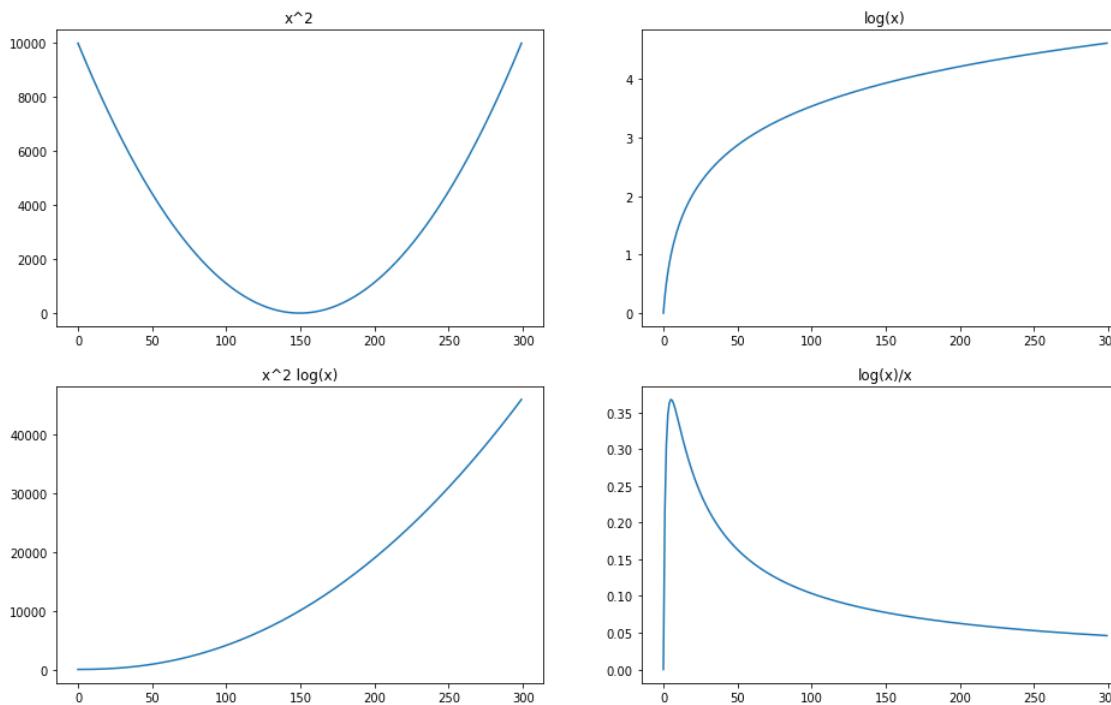
3.2 Subplot

Spesso può essere utile confrontare diversi grafici. Per fare ciò, possiamo avvalerci della funzione subplot, che permette di costruire una "griglia di grafici".

Vediamo ad esempio come stampare le seguenti funzioni in una griglia 2×2 :

$$y = x^2, \quad y = \log(x), \quad y = x^2 \cdot \log(x), \quad y = \log(x)/x$$

```
In [194]: 1 plt.figure(figsize=(16,10)) #definisce una figura con una data dimensione (in pollici)
2 plt.subplot(2,2,1) #definisce una griglia 2x2 e seleziona la prima cella come cella corrente
3 x = np.linspace(-100,100,300)
4 plt.plot(x**2) #primo grafico
5 plt.title('x^2') #imposta un titolo per la cella corrente
6
7 plt.subplot(2,2,2) #sempre nella stessa griglia 2x2, seleziona la seconda cella
8 x = np.linspace(1,100,300)
9 plt.plot(np.log(x)) #secondo grafico
10 plt.title('log(x)') #imposta un titolo per la cella corrente
11
12 plt.subplot(2,2,3) #seleziona la terza cella
13 x = np.linspace(1,100,300)
14 plt.plot(x**2*np.log(x)) #terzo grafico
15 plt.title('x^2 log(x)') #imposta un titolo per la cella corrente
16
17 plt.subplot(2,2,4) #seleziona la quarta cella
18 x = np.linspace(1,100,300)
19 plt.plot(np.log(x)/x) #quarto grafico
20 plt.title('log(x)/x') #imposta un titolo per la cella corrente
21
22 plt.show() #mostra la figura
```

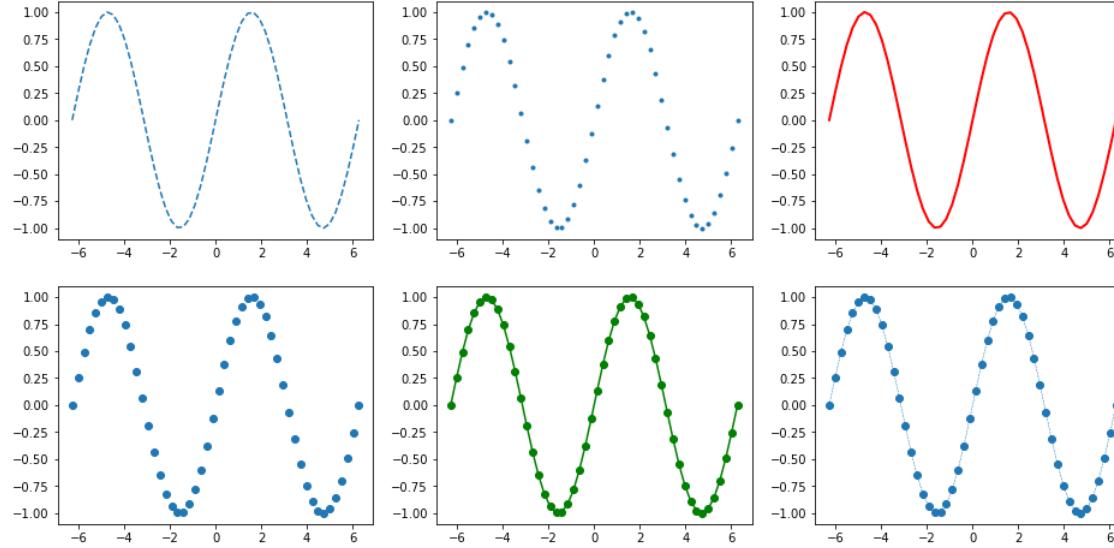


3.3 Colori e stili

E' possibile utilizzare diversi colori e stili per i grafici. Sotto alcuni esempi:

In [195]:

```
1 x = np.linspace(-2*np.pi,2*np.pi,50)
2 y = np.sin(x)
3
4 plt.figure(figsize=(16,8))
5 plt.subplot(231) #versione abbreviata di plt.subplot(2,3,1)
6 plt.plot(x,y,'--') #linea tratteggiata
7 plt.subplot(232)
8 plt.plot(x,y,'.') #solo punti
9 plt.subplot(233)
10 plt.plot(x,y,'r', linewidth=2) #linea rossa, spessore 2
11 plt.subplot(234)
12 plt.plot(x,y,'o') #cerchietti
13 plt.subplot(235)
14 plt.plot(x,y,'o-g') #cerchietti uniti da linee verdi
15 plt.subplot(236)
16 plt.plot(x,y,'o--', linewidth=0.5) #cerchietti uniti da linee tratteggiate, spessore 0.5
17
18 plt.show()
```

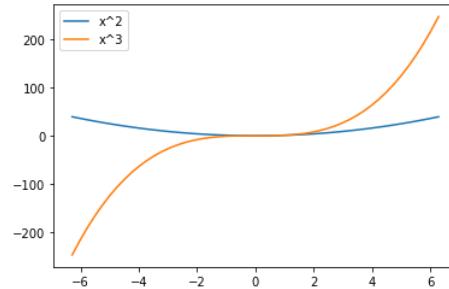


3.4 Sovrapposizione di linee e legenda

E' semplice sovrapporre più linee mediante `matplotlib`:

In [196]:

```
1 plt.figure()
2 plt.plot(x,x**2)
3 plt.plot(x,x**3) #al secondo plot, matplotlib cambierà automaticamente colore
4 plt.legend(['x^2','x^3']) #possiamo anche inserire una Legenda.
5 #Gli elementi della Legenda verranno associati alle linee nell'ordine in cui sono stati piazzati
6 plt.show()
```



Matplotlib è una libreria molto versatile. Sul sito ufficiale di Matplotlib è presente una galleria di esempi di plot con relativo codice (<https://matplotlib.org/2.1.1/gallery/index.html>). (<https://matplotlib.org/2.1.1/gallery/index.html>).

4 Pandas

Pandas è una libreria di alto livello che mette a disposizione diversi strumenti e strutture dati per l'analisi dei dati. In particolare, Pandas è molto utile per caricare, manipolare e visualizzare i dati in maniera veloce e conveniente prima di passare all'analisi vera e propria. Le due strutture dati principali di Pandas sono le `Series` e i `DataFrame`.

4.1 Series

Una `Series` è una struttura monodimensionale (una serie di dati) molto simile a un array di numpy. A differenza di esso, i suoi elementi possono essere indicizzati mediante delle etichette, oltre che mediante numeri. I valori contenuti in una serie possono essere di qualsiasi tipo.

Creazione di Series

E' possibile definire una `Series` a partire da una lista o da un array di numpy:

```
In [197]: 1 import pandas as pd
2 import numpy as np
3 np.random.seed(123) #impostiamo un seed per ripetibilità
4 s1 = pd.Series([7,5,2,8,9,6])# alternativamente print(pd.Series(np.array([7,5,2])))
5 print(s1)

0    7
1    5
2    2
3    8
4    9
5    6
dtype: int64
```

I numeri visualizzati sulla sinistra rappresentano le etichette dei valori contenute nella serie, che, di default sono di tipo numerico e sequenziale. Durante la definizione di una serie, è possibile specificare delle opportune etichette (una per ogni valore):

```
In [198]: 1 valori = [7,5,2,8,9,6]
2 etichette = ['a','b','c','d','e','f']
3 s2 = pd.Series(valori, index=etichette)
4 print(s2)

a    7
b    5
c    2
d    8
e    9
f    6
dtype: int64
```

E' possibile definire una serie anche mediante un dizionario che specifica contemporaneamente etichette e valori:

```
In [199]: 1 s3=pd.Series({'a':14,'h':18,'m':72})
2 print(s3)

a    14
h    18
m    72
dtype: int64
```

Opzionalmente, è possibile assegnare un nome a una serie:

```
In [200]: 1 pd.Series([1,2,3], name='Mia Serie')

Out[200]: 0    1
1    2
2    3
Name: Mia Serie, dtype: int64
```



Domanda 9

Qual è la principale differenza tra `Series` di Pandas e `array` di Numpy?



Risposta 9

Indicizzazione di Series

Quando gli indici sono numerici, le serie possono essere indicizzate come gli array di numpy:

```
In [201]: 1 print(s1[0]) #indicizzazione
2 print(s1[0:4:2]) #slicing

7
0    7
2    2
dtype: int64
```

Quando gli indici sono delle più generiche etichette, l'indicizzazione avviene in maniera simile, ma non è possibile utilizzare lo slicing:

```
In [202]: 1 print(s2['c'])

2
```

Chiaramente, è possibile modificare i valori di una serie mediante l'indicizzazione:

```
In [203]: 1 s2['c']=4  
2 s2
```

```
Out[203]: a    7  
b    5  
c    4  
d    8  
e    9  
f    6  
dtype: int64
```

Qualora l'indice specificato non esistesse, un nuovo elemento verrà creato:

```
In [204]: 1 s2['z']=-2  
2 s2
```

```
Out[204]: a    7  
b    5  
c    4  
d    8  
e    9  
f    6  
z   -2  
dtype: int64
```

Qualora volessimo specificare più di una etichetta alla volta, potremmo passare una lista di etichette:

```
In [205]: 1 print(s2[['a','c','d']])
```

```
a    7  
c    4  
d    8  
dtype: int64
```

Le serie con etichette alfanumeriche possono essere indicizzate anche seguendo l'ordine in cui i dati sono inseriti nella serie, in un certo senso "scartando" le etichette alfanumeriche e indicizzando gli elementi in maniera posizionale. Questo effetti si ottiene mediante il metodo `iloc`:

```
In [206]: 1 print(s2,'\n')  
2 print("Elemento di indice 'a':",s2['a'])  
3 print("Primo elemento della serie:",s2.iloc[0])
```

```
a    7  
b    5  
c    4  
d    8  
e    9  
f    6  
z   -2  
dtype: int64
```

```
Elemento di indice 'a': 7  
Primo elemento della serie: 7
```

In certi casi, può essere utile ripristinare la numerazione degli indici. Ciò può essere fatto usando il metodo `reset_index`:

```
In [207]: 1 print(s3,'\n')  
2 print(s3.reset_index(drop=True))#drop=True indica di scartare i vecchi indici
```

```
a    14  
h    18  
m    72  
dtype: int64
```



```
0    14  
1    18  
2    72  
dtype: int64
```

Le serie ammettono anche l'indicizzazione logica:

```
In [208]: 1 print(s1,'\n') #serie s1  
2 print(s1>2,'\\n') #indicizzazione logica per selezionare gli elementi maggiori di 2  
3 print(s1[s1>2],'\n') #applicazione dell'indicizzazione logica
```

```
0    7  
1    5  
2    2  
3    8  
4    9  
5    6  
dtype: int64
```



```
0    True  
1    True  
2    False  
3    True  
4    True  
5    True  
dtype: bool
```



```
0    7  
1    5  
3    8  
4    9  
5    6  
dtype: int64
```

E' possibile specificare la combinazione tra due condizione mediante gli operatori logici "`|`" (or) e "`&`" (and), ricordando di racchiudere gli operandi tra parentesi tonde:

```
In [209]: 1 (s1>2) & (s1<6)
```

```
Out[209]: 0 False
1 True
2 False
3 False
4 False
5 False
dtype: bool
```

```
In [210]: 1 s1[(s1>2) & (s1<6)]
```

```
Out[210]: 1 5
dtype: int64
```

Analogamente per l'or:

```
In [211]: 1 (s1<2) | (s1>6)
```

```
Out[211]: 0 True
1 False
2 False
3 True
4 True
5 False
dtype: bool
```

```
In [212]: 1 print(s1[(s1<2) | (s1>6)])
```

```
0 7
3 8
4 9
dtype: int64
```

Come nel caso degli array di Numpy, l'allocazione della memoria è gestita dinamicamente per le Serie. Pertanto, se assegno una serie ad una nuova variabile e modifco la seconda variabile, verrà modificata anche la prima:

```
In [213]: 1 s11=pd.Series([1,2,3])
2 s12=s11
3 s12[0]=-1
4 s11
```

```
Out[213]: 0 -1
1 2
2 3
dtype: int64
```

Per ottenere una nuova Serie indipendente, possiamo usare il metodo `copy`:

```
In [214]: 1 s11=pd.Series([1,2,3])
2 s12=s11.copy()
3 s12[0]=-1
4 s11
```

```
Out[214]: 0 1
1 2
2 3
dtype: int64
```

Domanda 10



Cosa stampa a schermo il seguente codice?

```
s=pd.Series([1,2,3,4,6])
print(s[s%2])
```

Risposta 10



Tipi di dato

Le `Series` possono contenere diversi tipi di dato:

```
In [215]: 1 pd.Series([2.5,3.4,5.2])
```

```
Out[215]: 0 2.5
1 3.4
2 5.2
dtype: float64
```

Ad una `Series` viene associato un unico tipo di dato. Se specifichiamo dati di tipi eterogenei, la `Series` sarà di tipo "object":

```
In [216]: s=pd.Series([2.5, 'A', 5.2])
```

```
Out[216]: 0    2.5
          1      A
          2    5.2
dtype: object
```

E' possibile cambiare il tipo di dato di una serie al volo con `astype` in maniera simile a quanto avviene con gli array di Numpy:

```
In [217]: s=pd.Series([2,3,8,9,12,45])
          print(s)
          print(s.astype(float))
```

```
0    2
1    3
2    8
3    9
4   12
5   45
dtype: int64
0    2.0
1    3.0
2    8.0
3    9.0
4   12.0
5   45.0
dtype: float64
```



Domanda 11

Considerata la Series definita come segue:

```
s=pd.Series([2.5, 'A', 5.2])
```

quali delle seguenti operazioni restituiscono un errore? Perche?

```
s.astype(str), s.astype(int)
```

Risposta 11



Operazioni su e tra Series

Sono definite sulle serie le principali operazioni disponibili per gli array di numpy:

```
In [218]: print("Min:",s1.min())
          2 print("Max:",s1.max())
          3 print("Mean:",s1.mean())
```

```
Min: 2
Max: 9
Mean: 6.166666666666667
```

E' possibile ottenere la dimensione di una serie mediante la funzione `len`:

```
In [219]: print(len(s1))
```

```
6
```

E' possibile ottenere delle statistiche sui valori univoci (ovvero i valori della serie, una volta rimossi i duplicati) contenuti in una serie mediante il metodo `unique`:

```
In [220]: print("Unique:",s1.unique()) #restituisce i valori univoci
```

```
Unique: [7 5 2 8 9 6]
```

Per conoscere il numero di valori univoci in una serie, possiamo utilizzare il metodo `nunique`:

```
In [221]: s1.nunique()
```

```
Out[221]: 6
```

E' possibile ottenere i valori univoci di una serie insieme alle frequenze con le quali essi appaiono nella serie mediante il metodo `value_counts`:

```
In [222]: 1 tmp = pd.Series(np.random.randint(0,10,100))
2 print(tmp.unique()) #valori univoci
3 tmp.value_counts() #valori univoci con relative frequenze
```

```
[2 6 1 3 9 0 4 7 8 5]
```

```
Out[222]: 3    15
6    14
1    13
9    10
4    10
2    10
0    10
7    8
8    5
5    5
dtype: int64
```

Il risultato di `value_counts` è una `Series` in cui gli indici rappresentano i valori univoci, mentre i valori sono le frequenze con cui essi appaiono nella serie. La serie è ordinata per valori.

Il metodo `describe` permette di calcolare diverse statistiche dei valori contenuti nella serie:

```
In [223]: 1 tmp.describe()
```

```
Out[223]: count    100.000000
mean      4.130000
std       2.830765
min       0.000000
25%      2.000000
50%      4.000000
75%      6.000000
max      9.000000
dtype: float64
```

Nelle operazioni tra serie, gli elementi vengono associati in base agli indici. Nel caso in cui esiste una perfetta corrispondenza tra gli elementi otteniamo il seguente risultato:

```
In [224]: 1 print(pd.Series([1,2,3])+pd.Series([4,4,4]), '\n')
2 print(pd.Series([1,2,3])*pd.Series([4,4,4]), '\n')
```

```
0    5
1    6
2    7
dtype: int64

0    4
1    8
2   12
dtype: int64
```

Nel caso in cui alcuni indici dovessero essere mancanti, le caselle corrispondenti verranno riempite con `NaN` (not a number) per indicare l'assenza del valore:

```
In [225]: 1 s1 = pd.Series([1,4,2], index = [1,2,3])
2 s2 = pd.Series([4,2,8], index = [0,1,2])
3 print(s1, '\n')
4 print(s2, '\n')
5 print(s1+s2)
```

```
1    1
2    4
3    2
dtype: int64

0    4
1    2
2    8
dtype: int64

0    NaN
1    3.0
2   12.0
3    NaN
dtype: float64
```

In questo caso, l'indice `0` era presente solo nella seconda serie (`s2`), mentre l'indice `3` era presente solo nella prima serie (`s1`).

Se vogliamo escludere i valori `NaN` (inclusi i relativi indici) possiamo utilizzare il metodo `dropna`:

```
In [226]: 1 s3=s1+s2
2 print(s3)
3 print(s3.dropna())
```

```
0    NaN
1    3.0
2   12.0
3    NaN
dtype: float64
1    3.0
2   12.0
dtype: float64
```

E' possibile applicare una funzione a tutti gli elementi di una serie mediante il metodo `apply`. Supponiamo ad esempio di voler trasformare delle stringhe contenute in una serie in uppercase. Possiamo specificare la funzione `str.upper` mediante il metodo `apply`:

```
In [227]: 1 s=pd.Series(['aba','cda','daf','acc'])
2 s.apply(str.upper)
```

```
Out[227]: 0    ABA
1    CDA
2    DAF
3    ACC
dtype: object
```

Mediane apply possiamo applicare anche funzioni definite dall'utente mediante espressioni lambda o mediante la consueta sintassi:

```
In [228]: 1 def miafun(x):
2     y="Stringa: "
3     return y+x
4 s.apply(miafun)
```

```
Out[228]: 0    Stringa: aba
1    Stringa: cda
2    Stringa: daf
3    Stringa: acc
dtype: object
```

La stessa funzione si può scrivere in maniera più compatta come espressione lambda:

```
In [229]: 1 s.apply(lambda x: "Stringa: "+x)
```

```
Out[229]: 0    Stringa: aba
1    Stringa: cda
2    Stringa: daf
3    Stringa: acc
dtype: object
```

E' possibile modificare tutte le occorrenze di un dato valore di una serie mediante il metodo replace :

```
In [230]: 1 ser = pd.Series([1,2,15,-1,7,9,2,-1])
2 print(ser)
3 ser=ser.replace({-1:0}) #sostituisce tutti le occorrenze di "-1" con zeri
4 print(ser)
```

```
0    1
1    2
2   15
3   -1
4    7
5    9
6    2
7   -1
dtype: int64
0    1
1    2
2   15
3    0
4    7
5    9
6    2
7    0
dtype: int64
```

Domanda 12



Si provi ad applicare il metodo apply ad una Series specificando una funzione che prende in input due argomenti. E' possibile farlo? Perché?

Risposta 12



Conversione in array di Numpy

E' possibile accedere ai valori della series in forma di array di numpy mediante la proprietà values :

```
In [231]: 1 print(s3.values)
```

```
[nan  3. 12. nan]
```

Bisogna fare attenzione al fatto che values non crea una copia indipendente della serie, per cui, se modifichiamo l'array di numpy accessibile mediante values , di fatto modifichiamo anche la serie:

```
In [232]:
```

```
1 a = s3.values
2 print(s3)
3 a[0]=-1
4 print(s3)
```

```
0      NaN
1      3.0
2     12.0
3      NaN
dtype: float64
0     -1.0
1      3.0
2     12.0
3      NaN
dtype: float64
```

Domanda 13



Come si può ottenere un array di Numpy da un Series in modo che le due entità siano indipendenti?

Risposta 13



4.2 DataFrame *

Un **DataFrame** è sostanzialmente una tabella di numeri in cui:

- Ogni riga rappresenta una **osservazione** diversa;
- Ogni colonna rappresenta una variabile.

Righe e colonne possono avere dei nomi. In particolare, è molto comune assegnare dei nomi alle colonne per indicare a quale variabile corrisponde ognuna di esse.

Costruzione e visualizzazione di DataFrame *

E' possibile costruire un DataFrame a partire da un array bidimensionale (una matrice) di numpy:

```
In [233]:
```

```
1 dati = np.random.rand(10,3) #matrice di valori random 10 x 3
2 #si tratta di una matrice di 10 osservazioni, ognuna caratterizzata da 3 variabili
3
4 df = pd.DataFrame(dati,columns=['A','B','C'])
5
6 df #in jupyter o in una shell ipython possiamo stampare il dataframe
7 #semplicemente scrivendo "df". In uno script dovremmo scrivere "print df"
```

Out[233]:

	A	B	C
0	0.309884	0.507204	0.280793
1	0.763837	0.108542	0.511655
2	0.909769	0.218376	0.363104
3	0.854973	0.711392	0.392944
4	0.231301	0.380175	0.549162
5	0.556719	0.004135	0.638023
6	0.057648	0.043027	0.875051
7	0.292588	0.762768	0.367865
8	0.873502	0.029424	0.552044
9	0.240248	0.884805	0.460238

Ad ogni riga è stato assegnato in automatico un indice numerico. Se vogliamo possiamo specificare nomi anche per le righe:

```
In [234]:
```

```
1 np.random.seed(123)#impostiamo un seed per ripetitibilità
2 df = pd.DataFrame(np.random.rand(4,3),columns=['A','B','C'],index=['X','Y','Z','W'])
```

Out[234]:

	A	B	C
X	0.696469	0.286139	0.226851
Y	0.551315	0.719469	0.423106
Z	0.980764	0.684830	0.480932
W	0.392118	0.343178	0.729050

Analogamente a quanto visto nel caso delle serie, è possibile costruire un DataFrame mediante un dizionario che specifica nome e valori di ogni colonna:

```
In [235]: 1 pd.DataFrame({'A':np.random.rand(10), 'B':np.random.rand(10), 'C':np.random.rand(10)})
```

Out[235]:

	A	B	C
0	0.438572	0.724455	0.430863
1	0.059678	0.611024	0.493685
2	0.398044	0.722443	0.425830
3	0.737995	0.322959	0.312261
4	0.182492	0.361789	0.426351
5	0.175452	0.228263	0.893389
6	0.531551	0.293714	0.944160
7	0.531828	0.630976	0.501837
8	0.634401	0.092105	0.623953
9	0.849432	0.433701	0.115618

Nel caso di dataset molto grandi, possiamo visualizzare solo le prime righe usando il metodo **head**:

```
In [236]: 1 df_big = pd.DataFrame(np.random.rand(100,3),columns=['A','B','C'])
2 df_big.head()
```

Out[236]:

	A	B	C
0	0.317285	0.414826	0.866309
1	0.250455	0.483034	0.985560
2	0.519485	0.612895	0.120629
3	0.826341	0.603060	0.545068
4	0.342764	0.304121	0.417022

E' anche possibile specificare il numero di righe da mostrare:

```
In [237]: 1 df_big.head(2)
```

Out[237]:

	A	B	C
0	0.317285	0.414826	0.866309
1	0.250455	0.483034	0.985560

In maniera simile, possiamo mostrare le ultime righe con **tail**:

```
In [238]: 1 df_big.tail(3)
```

Out[238]:

	A	B	C
97	0.811953	0.335544	0.349566
98	0.389874	0.754797	0.369291
99	0.242220	0.937668	0.908011

Possiamo ottenere il numero di righe del **DataFrame** mediante la funzione **len**:

```
In [239]: 1 print(len(df_big))
100
```

Se vogliamo conoscere sia il numero di righe che il numero di colonne, possiamo richiamare la proprietà **shape**:

```
In [240]: 1 print(df_big.shape)
(100, 3)
```

Analogamente a quanto visto per le Series, possiamo visualizzare un DataFrame come un array di numpy richiamando la proprietà **values** :

```
In [241]: 1 print(df,'\\n')
2 print(df.values)

A          B          C
X  0.696469  0.286139  0.226851
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050

[[0.69646919  0.28613933  0.22685145]
 [0.55131477  0.71946897  0.42310646]
 [0.9807642   0.68482974  0.4809319 ]
 [0.39211752  0.34317802  0.72904971]]
```

Anche per i DataFrame valgono le stesse considerazioni sulla memoria fatte per le Series. Per ottenere una copia indipendente di una DataFrame, è possibile utilizzare il metodo **copy** :

```
In [242]: 1 df2=df.copy()
```

Domanda 14

In cosa differisce principalmente un DataFrame da un array bidimensionale di Numpy? Qual è il vantaggio dei DataFrame?



Risposta 14



Indicizzazione *

Pandas mette a disposizione una serie di strumenti per indicizzare i DataFrame selezionandone righe o colonne. Ad esempio, possiamo selezionare la colonna B come segue:

```
In [243]: 1 s=df['B']
2 print(s,'\n')
3 print("Tipo di s:",type(s))

X 0.286139
Y 0.719469
Z 0.684830
W 0.343178
Name: B, dtype: float64

Tipo di s: <class 'pandas.core.series.Series'>
```

Da notare che il risultato di questa operazione è una Series (un DataFrame è in fondo una collezione di Series, ognuna rappresentante una colonna) che ha come nome il nome della colonna considerata. Possiamo selezionare più di una riga specificando una lista di righe:

```
In [244]: 1 dfAB = df[['A','C']]
2 dfAB

Out[244]:
      A      C
X  0.696469  0.226851
Y  0.551315  0.423106
Z  0.980764  0.480932
W  0.392118  0.729050
```

Il risultato di questa operazione è invece un DataFrame. La selezione delle righe avviene mediante la proprietà loc :

```
In [245]: 1 df5 = df.loc['X']
2 df5

Out[245]: A    0.696469
            B    0.286139
            C    0.226851
Name: X, dtype: float64
```

Anche il risultato di questa operazione è una Series, ma in questo caso gli indici rappresentano i nomi delle colonne, mentre il nome della serie corrisponde all'indice della riga selezionata. Come nel caso delle Series, possiamo usare iloc per indicizzare le righe in maniera posizionale:

```
In [246]: 1 df.iloc[1] #equivalente a df.loc['Y']

Out[246]: A    0.551315
            B    0.719469
            C    0.423106
Name: Y, dtype: float64
```

E' anche possibile concatenare le operazioni di indicizzazione per selezionare uno specifico valore:

```
In [247]: 1 print(df.iloc[1]['A'])
2 print(df['A'].iloc[1])

0.5513147690828912
0.5513147690828912
```

Anche in questo caso possiamo utilizzare l'indicizzazione logica in maniera simile a quanto visto per numpy:

```
In [248]: 1 df_big2=df_big[df_big['C']>0.5]
2 print(len(df_big), len(df_big2))
3 df_big2.head() #alcuni indici sono mancanti in quanto le righe corrispondenti sono state rimosse

100 53
```

```
Out[248]:
      A      B      C
0  0.317285  0.414826  0.866309
1  0.250455  0.483034  0.985560
3  0.826341  0.603060  0.545068
5  0.681301  0.875457  0.510422
6  0.669314  0.585937  0.624904
```

E' possibile combinare quanto visto finora per manipolare i dati in maniera semplice e veloce. Supponiamo di voler selezionare le righe per le quali la somma tra i valori di B e C è minore di 0.7 e supponiamo di essere interessati solo ai valori A di tali righe (e non all'intera riga di valori A,B,C). Il risultato che ci aspettiamo è un array monodimensionale di valori. Possiamo ottenere il risultato voluto come segue:

```
In [249]: 1 res = df[(df['B']+df['C'])>0.7]['A']
2 print(res.head(), res.shape)

Y 0.551315
Z 0.980764
W 0.392118
Name: A, dtype: float64 (3,)
```

Possiamo applicare l'indicizzazione anche a livello dell'intera tabella (oltre che al livello delle singole colonne):

```
In [250]: 1 df>0.3
```

```
Out[250]:
A   B   C
X  True False False
Y  True  True  True
Z  True  True  True
W  True  True  True
```

Se applichiamo questa indicizzazione al DataFrame otterremo l'apparizione di alcuni NaN, che indicano la presenza degli elementi che non rispettano la condizione considerata:

```
In [251]: 1 df[df>0.3]
```

```
Out[251]:
A   B   C
X  0.696469  NaN  NaN
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050
```

Possiamo rimuovere i valori NaN mediante il metodo dropna come visto nel caso delle Series. Tuttavia, in questo caso, verranno rimosse tutte le righe che presentano almeno un NaN :

```
In [252]: 1 df[df>0.3].dropna()
```

```
Out[252]:
A   B   C
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050
```

Possiamo chiedere a dropna di rimuovere le colonne che presentano almeno un NaN specificando axis=1 :

```
In [253]: 1 df[df>0.3].dropna(axis=1)
```

```
Out[253]:
A
X  0.696469
Y  0.551315
Z  0.980764
W  0.392118
```

Alternativamente possiamo sostituire i valori NaN al volo mediante la funzione fillna :

```
In [254]: 1 df[df>0.3].fillna('VAL') #sostituisce i NaN con 'VAL'
```

```
Out[254]:
A   B   C
X  0.696469  VAL  VAL
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050
```

Anche in questo caso, come visto nel caso delle Series possiamo ripristinare l'ordine degli indici mediante il metodo reset_index :

```
In [255]: 1 print(df)
2 print(df.reset_index(drop=True))
```

```
A   B   C
X  0.696469  0.286139  0.226851
Y  0.551315  0.719469  0.423106
Z  0.980764  0.684830  0.480932
W  0.392118  0.343178  0.729050
A   B   C
0  0.696469  0.286139  0.226851
1  0.551315  0.719469  0.423106
2  0.980764  0.684830  0.480932
3  0.392118  0.343178  0.729050
```

Se non specifichiamo drop=True, i vecchi indici verranno mantenuti come nuova colonna:

```
In [256]: 1 print(df.reset_index())
```

index	A	B	C
0	X 0.696469	0.286139	0.226851
1	Y 0.551315	0.719469	0.423106
2	Z 0.980764	0.684830	0.480932
3	W 0.392118	0.343178	0.729050

Possiamo impostare qualsiasi colonna come nuovo indice. Ad esempio:

```
In [257]: 1 df.set_index('A')
```

```
Out[257]:
```

B	C
A	
0.696469	0.286139 0.226851
0.551315	0.719469 0.423106
0.980764	0.684830 0.480932
0.392118	0.343178 0.729050

Va notato che questa operazione non modifica di fatto il DataFrame, ma crea una nuova "vista" dei dati con la modifica richiesta:

```
In [258]: 1 df
```

```
Out[258]:
```

A	B	C
X 0.696469	0.286139	0.226851
Y 0.551315	0.719469	0.423106
Z 0.980764	0.684830	0.480932
W 0.392118	0.343178	0.729050

Per utilizzare questa versione modificata del dataframe, possiamo salvarla in un'altra variabile:

```
In [259]: 1 df2=df.set_index('A')  
2 df2
```

```
Out[259]:
```

B	C
A	
0.696469	0.286139 0.226851
0.551315	0.719469 0.423106
0.980764	0.684830 0.480932
0.392118	0.343178 0.729050

Domanda 15



Si consideri il seguente DataFrame:

```
pd.DataFrame({'A':[1,2,3,4], 'B':[3,2,6,7], 'C':[0,2,-1,12]})
```

Si utilizzi l'indicizzazione logica per selezionare le righe in cui la somma dei valori di 'C' ed 'A' sia maggiore di 1.

Risposta 15



Manipolazione di DataFrame *

I valori contenuti nelle righe e nelle colonne del dataframe possono essere facilmente modificati. Il seguente esempio moltiplica tutti i valori della colonna B per 2:

```
In [260]: 1 df['B']*=2  
2 df.head()
```

```
Out[260]:
```

A	B	C
X 0.696469	0.572279	0.226851
Y 0.551315	1.438938	0.423106
Z 0.980764	1.369659	0.480932
W 0.392118	0.686356	0.729050

Analogamente possiamo dividere tutti i valori della riga di indice 2 per 3:

```
In [261]: 1 df.iloc[2]=df.iloc[2]/3  
2 df.head()
```

Out[261]:

	A	B	C
X	0.696469	0.572279	0.226851
Y	0.551315	1.438938	0.423106
Z	0.326921	0.456553	0.160311
W	0.392118	0.686356	0.729050

Possiamo definire una nuova colonna con una semplice operazione di assegnamento:

```
In [262]: 1 df['D']=df['A']+df['C']  
2 df['E']=np.ones(len(df))*5  
3 df.head()
```

Out[262]:

	A	B	C	D	E
X	0.696469	0.572279	0.226851	0.923321	5.0
Y	0.551315	1.438938	0.423106	0.974421	5.0
Z	0.326921	0.456553	0.160311	0.487232	5.0
W	0.392118	0.686356	0.729050	1.121167	5.0

Possiamo rimuovere una colonna mediante il metodo `drop` e specificando `axis=1` per indicare che vogliamo rimuovere una colonna:

```
In [263]: 1 df.drop('E',axis=1).head()
```

Out[263]:

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

Il metodo `drop` non modifica il DataFrame ma genera solo una nuova "vista" senza la colonna da rimuovere:

```
In [264]: 1 df.head()
```

Out[264]:

	A	B	C	D	E
X	0.696469	0.572279	0.226851	0.923321	5.0
Y	0.551315	1.438938	0.423106	0.974421	5.0
Z	0.326921	0.456553	0.160311	0.487232	5.0
W	0.392118	0.686356	0.729050	1.121167	5.0

Possiamo rimuovere la colonna effettivamente mediante un assegnamento:

```
In [265]: 1 df=df.drop('E',axis=1)  
2 df.head()
```

Out[265]:

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

La rimozione delle righe avviene allo stesso modo, ma bisogna specificare `axis =0`:

```
In [266]: 1 df.drop('X', axis=0)
```

Out[266]:

	A	B	C	D
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167

E' inoltre possibile aggiungere una nuova riga in coda al DataFrame mediante il metodo `append`. Dato che le righe di un DataFrame sono delle Series, dovremo costruire una serie con i giusti indici (corrispondenti alle colonne del DataFrame) e il giusto nome (corrispondente al nuovo indice):

```
In [267]: 1 new_row=pd.Series([1,2,3,4], index=['A','B','C','D'], name='H')
2 print(new_row)
3 df.append(new_row)
```

```
A 1
B 2
C 3
D 4
Name: H, dtype: int64
```

Out[267]:

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167
H	1.000000	2.000000	3.000000	4.000000

Possiamo aggiungere più di una riga alla volta specificando un DataFrame:

```
In [268]: 1 new_rows = pd.DataFrame({'A':[0,1],'B':[2,3],'C':[4,5],'D':[6,7]}, index=['H','K'])
2 new_rows
```

Out[268]:

	A	B	C	D
H	0	2	4	6
K	1	3	5	7

```
In [269]: 1 df.append(new_rows)
```

Out[269]:

	A	B	C	D
X	0.696469	0.572279	0.226851	0.923321
Y	0.551315	1.438938	0.423106	0.974421
Z	0.326921	0.456553	0.160311	0.487232
W	0.392118	0.686356	0.729050	1.121167
H	0.000000	2.000000	4.000000	6.000000
K	1.000000	3.000000	5.000000	7.000000

Domanda 16



Si consideri il seguente DataFrame:

```
pd.DataFrame({'A':[1,2,3,4],'B':[3,2,6,7],'C':[0,2,-1,12]})
```

Si inserisca nel DataFrame una nuova colonna 'D' che contenga il valore 1 in tutte le righe in cui il valore di B è superiore al valore di C.

Suggerimento: si usi "astype(int)" per trasformare i booleani in interi.

Risposta 16



Operazioni tra e su DataFrame *

Restano definite sui DataFrame, con le opportune differenze, le operazioni viste nel caso delle serie. In genere, queste vengono applicate a tutte le colonne del DataFrame in maniera indipendente:

```
In [270]: 1 df.mean() #media di ogni colonna
```

```
Out[270]: A    0.491706
          B    0.788531
          C    0.384830
          D    0.876535
          dtype: float64
```

```
In [271]: 1 df.max() #massimo di ogni colonna
```

```
Out[271]: A    0.696469
          B    1.438938
          C    0.729050
          D    1.121167
          dtype: float64
```

In [272]: 1 df.describe() #statistiche per ogni colonna

Out[272]:

	A	B	C	D
count	4.000000	4.000000	4.000000	4.000000
mean	0.491706	0.788531	0.384830	0.876535
std	0.165884	0.443638	0.255159	0.272747
min	0.326921	0.456553	0.160311	0.487232
25%	0.375818	0.543347	0.210216	0.814298
50%	0.471716	0.629317	0.324979	0.948871
75%	0.587603	0.874502	0.499592	1.011108
max	0.696469	1.438938	0.729050	1.121167

Dato che le colonne di un DataFrame sono delle serie, ad esse può essere applicato il metodo apply :

In [273]: 1 df['A']=df['A'].apply(lambda x: "Numero: "+str(x))
2 df

Out[273]:

	A	B	C	D
X	Numero: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Numero: 0.5513147690828912	1.438938	0.423106	0.974421
Z	Numero: 0.3269213994615385	0.456553	0.160311	0.487232
W	Numero: 0.3921175181941505	0.686356	0.729050	1.121167

E' possibile ordinare le righe di un DataFrame rispetto ai valori di una delle colonne mediante il metodo sort_values :

In [274]: 1 df.sort_values(by='D')

Out[274]:

	A	B	C	D
Z	Numero: 0.3269213994615385	0.456553	0.160311	0.487232
X	Numero: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Numero: 0.5513147690828912	1.438938	0.423106	0.974421
W	Numero: 0.3921175181941505	0.686356	0.729050	1.121167

Per rendere l'ordinamento permanente, dobbiamo effettuare un assegnamento:

In [275]: 1 df=df.sort_values(by='D')
2 df

Out[275]:

	A	B	C	D
Z	Numero: 0.3269213994615385	0.456553	0.160311	0.487232
X	Numero: 0.6964691855978616	0.572279	0.226851	0.923321
Y	Numero: 0.5513147690828912	1.438938	0.423106	0.974421
W	Numero: 0.3921175181941505	0.686356	0.729050	1.121167

Domanda 17



Si consideri il seguente DataFrame:

```
pd.DataFrame({'A':[1,2,3,4], 'B':[3,2,6,7], 'C':[0,2,-1,12]})
```

Si trasformi la colonna "B" in modo che i nuovi valori siano:

- uguali a zero se precedentemente pari;
- uguali a -1 se precedentemente dispari.

Risposta 17



Groupby *

Il metodo groupby permette di raggruppare le righe di un DataFrame e richiamare delle funzioni aggregate su di esse. Consideriamo un DataFrame un po' più rappresentativo:

```
In [276]: 1 df=pd.DataFrame({'income':[10000,11000,9000,3000,1000,5000,7000,2000,7000,12000,8000],\
2           'age':[32,32,45,35,28,18,27,45,39,33,32],\
3           'sex':['M','F','M','M','F','M','M','F','F'],\
4           'company':['CDX','FLZ','PTX','CDX','PTX','CDX','FLZ','CDX','FLZ','PTX','FLZ']})
```

Out[276]:

	income	age	sex	company
0	10000	32	M	CDX
1	11000	32	F	FLZ
2	9000	45	M	PTX
3	3000	35	M	CDX
4	1000	28	M	PTX
5	5000	18	F	CDX
6	7000	27	F	FLZ
7	2000	45	M	CDX
8	7000	39	M	FLZ
9	12000	33	F	PTX
10	8000	32	F	FLZ

Il metodo `groupby` ci permette di raggruppare le righe del `DataFrame` per valore, rispetto a una colonna specificata. Supponiamo di voler raggruppare tutte le righe che hanno lo stesso valore di `sex`:

```
In [277]: 1 df.groupby('sex')
```

Out[277]: <pandas.core.groupby.generic.DataFrameGroupBy object at 0x7ffbbbe17940>

Questa operazione restituisce un oggetto di tipo `DataFrameGroupBy` sul quale sarà possibile effettuare delle operazioni aggregate (ad esempio, somme e medie). Supponiamo adesso di voler calcolare la media di tutti i valori che ricadono nello stesso gruppo (ovvero calcoliamo la media delle righe che hanno lo stesso valore di `sex`):

```
In [278]: 1 df.groupby('sex').mean()
```

Out[278]:

	income	age
sex		
F	8600.000000	28.400000
M	5333.333333	37.333333

Se siamo interessati a una sola delle variabili, possiamo selezionarla prima o dopo l'operazione sui dati aggregati:

```
In [279]: 1 df.groupby('sex')['age'].mean() #equivolentemente: df.groupby('sex').mean()['age']
```

Out[279]:

sex	age
F	28.400000
M	37.333333

Name: age, dtype: float64

La tabella mostra il reddito medio e l'età media dei soggetti di sesso maschile e femminile. Dato che l'operazione di media si può applicare solo a valori numerici, la colonna Company è stata esclusa. Possiamo ottenere una tabella simile in cui mostriamo la somma dei redditi e la somma delle età cambiando `mean` in `sum`:

```
In [280]: 1 df.groupby('sex').sum()
```

Out[280]:

	income	age
sex		
F	43000	142
M	32000	224

In generale, è possibile utilizzare diverse funzioni aggregate oltre a `mean` e `sum`. Alcuni esempi sono `min`, `max`, `std`. Due funzioni particolarmente interessanti da usare in questo contesto sono `count` e `describe`. In particolare, `count` conta il numero di elementi interessati, mentre `describe` calcola diverse statistiche dei valori interessati. Vediamo due esempi:

```
In [281]: 1 df.groupby('sex').count()
```

Out[281]:

	income	age	company
sex			
F	5	5	5
M	6	6	6

Il numero di elementi è uguale per le varie colonne in quanto non ci sono valori `NaN` nel DataFrame.

```
In [282]: 1 df.groupby('sex').describe()
```

Out[282]:

	income					age										
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
sex																
F	5.0	8600.000000	2880.972058	5000.0	7000.0	8000.0	11000.0	12000.0	5.0	28.400000	6.268971	18.0	27.00	32.0	32.0	33.0
M	6.0	5333.333333	3829.708431	1000.0	2250.0	5000.0	8500.0	10000.0	6.0	37.333333	6.947422	28.0	32.75	37.0	43.5	45.0

Per ogni variabile numerica (age e income) sono state calcolate diverse statistiche. A volte può essere più chiaro visualizzare il dataframe trasposto:

```
In [283]: 1 df.groupby('sex').describe().transpose()
```

Out[283]:

	sex	F	M
income	count	5.000000	6.000000
	mean	8600.000000	5333.333333
	std	2880.972058	3829.708431
	min	5000.000000	1000.000000
	25%	7000.000000	2250.000000
	50%	8000.000000	5000.000000
	75%	11000.000000	8500.000000
	max	12000.000000	10000.000000
age	count	5.000000	6.000000
	mean	28.400000	37.333333
	std	6.268971	6.947422
	min	18.000000	28.000000
	25%	27.000000	32.750000
	50%	32.000000	37.000000
	75%	32.000000	43.500000
	max	33.000000	45.000000

Questa vista ci permette di comparare diverse statistiche delle due variabili age e income per M e F.

Domanda 18



Considerando il DataFrame precedentemente creato, si usi groupby per ottenere la somma dei redditi dei dipendenti di una data compagnia.

Risposta 18



Crosstab

Le Crosstab permettono di descrivere le relazioni tra due o più variabili **categoriche**. Una volta specificata una coppia di variabili categoriche, le righe e colonne della crosstab (nota anche come "tabella di contingenza") enumerano indipendentemente tutti i valori univoci delle due variabili categoriche, così che ogni cella della crosstab identifica una determinata coppia di volari. All'interno delle celle, vengono dunque riportati i numeri di elementi per i quali le due variabili categoriche assumono una determinata coppia di valori.

Supponiamo di voler studiare le relazioni tra company e sex :

```
In [284]: 1 pd.crosstab(df['sex'],df['company'])
```

Out[284]:

company	CDX	FLZ	PTX
sex			
F	1	3	1
M	3	1	2

La tabella sopra ci dice ad esempio che nella compagnia CDX, 1 soggetto è di sesso femminile, mentre 3 soggetti sono di sesso maschile. Allo stesso modo, un soggetto della compagnia FLZ è di sesso maschile, mentre tre soggetti sono di sesso femminile. E' possibile ottenere delle frequenze invece che dei conteggi, mediante normalize=True :

```
In [285]: 1 pd.crosstab(df['sex'],df['company'], normalize=True)
```

Out[285]:

company	CDX	FLZ	PTX
sex			
F	0.090909	0.272727	0.090909
M	0.272727	0.090909	0.181818

Alternativamente, possiamo normalizzare la tabella solo per righe o solo per colonne specificando normalize='index' o normalize='columns' :

```
In [286]: 1 pd.crosstab(df['sex'],df['company'], normalize='index')
```

Out[286]:

company	CDX	FLZ	PTX
sex			
F	0.2	0.600000	0.200000
M	0.5	0.166667	0.333333

Questa tabella riporta le percentuali di persone che lavorano nelle tre diverse compagnie per ciascun sesso, es., "il 20% delle donne lavora presso CDX". Analogamente possiamo normalizzare per colonne come segue:

```
In [287]: 1 pd.crosstab(df['sex'],df['company'], normalize='columns')
```

```
Out[287]:
```

sex	CDX	FLZ	PTX
F	0.25	0.75	0.333333
M	0.75	0.25	0.666667

Questa tabella riporta le percentuali di uomini e donne che lavorano in ciascuna compagnia, es., "il 25% dei lavoratori di CDX sono donne".

Se vogliamo studiare le relazioni tra più di due variabili categoriche, possiamo specificare una lista di colonne quando costruiamo la crosstab:

```
In [288]: 1 pd.crosstab(df['sex'],[df['age'],df['company']])
```

```
Out[288]:
```

age	18	27	28	32	33	35	39	45		
company	CDX	FLZ	PTX	CDX	FLZ	PTX	CDX	FLZ	CDX	PTX
sex										
F	1	1	0	0	2	1	0	0	0	0
M	0	0	1	1	0	0	1	1	1	1

Ogni cella della crosstab sopra conta il numero di osservazioni che riportano una determinata terna di valori. Ad esempio, 1 soggetto è di sesso maschile, lavora per PTX e ha 28 anni, mentre 2 soggetti sono di sesso femminile, lavorano per FLZ e hanno 32 anni.

Oltre a riportare conteggi e frequenze, una crosstab permette di calcolare statistiche di terze variabili considerate non categoriche. Supponiamo di voler conoscere l'età media delle persone di un dato sesso che lavorano in una data azienda. Possiamo costruire una crosstab specificando una nuova variabile (age) per values. Dato che di questa variabile bisognerà calcolare un qualche valore aggregato, dobbiamo anche specificare aggfunc per esempio pari a mean (per calcolare la media dei valori interessati):

```
In [289]: 1 pd.crosstab(df['sex'],df['company'], values=df['age'], aggfunc='mean')
```

```
Out[289]:
```

company	CDX	FLZ	PTX
sex			
F	18.000000	30.333333	33.0
M	37.333333	39.000000	36.5

La tabella ci dice che l'età media delle persone di sesso maschile che lavorano per CDX è di 37,33 anni.



Domanda 19

Si costruisca una crosstab che, per ogni compagnia, riporti il numero di dipendenti di una data età.

Risposta 19



Manipolazione "esplicita" di DataFrame

In alcuni casi può essere utile trattare i DataFrame "esplicitamente" come matrici di valori. Consideriamo ad esempio il seguente DataFrame:

```
In [290]: 1 df123 = pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,1,2]})  
2 df123
```

```
Out[290]:
```

Category	NumberOfElements
0	1
1	2
2	3

Supponiamo di voler costruire un nuovo DataFrame che, per ogni riga del dataframe df123 contenga esattamente "NumberOfElements" righe con valore di "NumberOfElements" pari a uno. Vogliamo in pratica "espandere" il DataFrame sopra come segue:

```
In [291]: 1 df123 = pd.DataFrame({'Category':[1,1,1,2,3,3], 'NumberOfElements':[1,1,1,1,1,1]})  
2 df123
```

```
Out[291]:
```

Category	NumberOfElements
0	1
1	1
2	1
3	1
4	1
5	1

Per farlo in maniera automatica, possiamo trattare il DataFrame più "esplicitamente" come una matrice di valori iterandone le righe. Il nuovo DataFrame sarà dapprima costruito come una lista di Series (le righe del DataFrame) e poi trasformato in un DataFrame:

```
In [292]: 1 newdat = []
2 for i, row in df123.iterrows(): #iterrows permette di iterare le righe di un DataFrame
3     for j in range(row['NumberOfElements']):
4         newrow = row.copy()
5         newrow['NumberOfElements']=1
6         newdat.append(newrow)
7 pd.DataFrame(newdat)
```

Out[292]:

	Category	NumberOfElements
0	1	1
1	1	1
2	1	1
3	2	1
4	3	1
5	3	1

```
In [293]: 1 pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,5,3], 'CheckedElements':[1,2,1]})
```

Out[293]:

	Category	NumberOfElements	CheckedElements
0	1	3	1
1	2	5	2
2	3	3	1

Input/Output

Pandas mette a disposizione diverse funzioni per leggere e scrivere dati. Vedremo in particolare le funzioni per leggere e scrivere file csv . E' possibile leggere file csv (https://it.wikipedia.org/wiki/Comma-separated_values) mediante la funzione pd.read_csv . La lettura può avvenire da file locali passando il path relativo o da URL:

```
In [294]: 1 data=pd.read_csv('http://iplab.dmi.unict.it/furnari/downloads/students.csv')
2 data.head()
```

Out[294]:

	admit	gre	gpa	prestige
0	0	380	3.61	3
1	1	660	3.67	3
2	1	800	4.00	1
3	1	640	3.19	4
4	0	520	2.93	4

Allo stesso modo, è possibile scrivere un DataFrame su file csv come segue:

```
In [295]: 1 data.to_csv('file.csv')
```

Dato che gli indici possono non essere sequenziali in un DataFrame, Pandas li inserisce nel csv come "colonna senza nome". Ad esempio, le prime righe del file che abbiamo appena scritto (file.csv) sono le seguenti:

```
,admit,gre,gpa,prestige
0,0,380,3.61,3
1,1,660,3.67,3
2,1,800,4.0,1
3,1,640,3.19,4
4,0,520,2.93,4
```

Come si può vedere, la prima colonna contiene i valori degli indici, ma non ha nome. Ciò può causare problemi quando carichiamo nuovamente il DataFrame:

```
In [296]: 1 data=pd.read_csv('file.csv')
2 data.head()
```

Out[296]:

	Unnamed: 0	admit	gre	gpa	prestige
0	0	0	380	3.61	3
1	1	1	660	3.67	3
2	2	1	800	4.00	1
3	3	1	640	3.19	4
4	4	0	520	2.93	4

Possiamo risolvere il problema di diversi modi:

- Eliminare la colonna "Unnamed: 0" dal DataFrame appena caricato (solo se gli indici sono sequenziali);
- Specificare di usare la colonna "Unnamed: 0" come colonna degli indici durante il caricamento;
- Salvare il DataFrame senza indici (solo se gli indici sono sequenziali).

Vediamo i tre casi:

```
In [297]: 1 data=pd.read_csv('file.csv')
2 data.drop('Unnamed: 0', axis=1).head()
```

```
Out[297]:
      admit  gre  gpa  prestige
0       0   380  3.61        3
1       1   660  3.67        3
2       1   800  4.00        1
3       1   640  3.19        4
4       0   520  2.93        4
```

```
In [298]: 1 data=pd.read_csv('file.csv', index_col='Unnamed: 0')
2 data.head()
```

```
Out[298]:
      admit  gre  gpa  prestige
0       0   380  3.61        3
1       1   660  3.67        3
2       1   800  4.00        1
3       1   640  3.19        4
4       0   520  2.93        4
```

```
In [299]: 1 data.to_csv('file.csv', index=False)
2 pd.read_csv('file.csv').head()
```

```
Out[299]:
      admit  gre  gpa  prestige
0       0   380  3.61        3
1       1   660  3.67        3
2       1   800  4.00        1
3       1   640  3.19        4
4       0   520  2.93        4
```

4.3 Esempio di manipolazione dati

Spesso, i dati devono essere opportunamente trattati per poterli analizzare. Vediamo un esempio di come combinare gli strumenti discussi sopra per manipolare un dataset reale.

Considereremo dataset disponibile a questo URL: <https://www.kaggle.com/lava18/google-play-store-apps>. Una copia del dataset è disponibile al seguente URL per scopi didattici: <http://iplab.dmi.unict.it/furnari/downloads/googleplaystore.csv>. Carichiamolo il dataset e visualizziamo le prime righe per assicurarcene che sia stato caricato correttamente:

```
In [300]: 1 data = pd.read_csv('http://iplab.dmi.unict.it/furnari/downloads/googleplaystore.csv')
2 data.head()
```

```
Out[300]:
          App    Category  Rating  Reviews  Size  Installs  Type  Price  Content Rating  Genres  Last Updated  Current Ver  Android Ver
0  Photo Editor & Candy Camera & Grid & ScrapBook  ART_AND_DESIGN    4.1     159   19M  10,000+   Free    0  Everyone  Art & Design  January 7, 2018      1.0.0    4.0.3 and up
1           Coloring book moana  ART_AND_DESIGN    3.9     967   14M  500,000+   Free    0  Everyone  Art & Design;Pretend Play  January 15, 2018      2.0.0    4.0.3 and up
2            U Launcher Lite – FREE Live Cool Themes, Hide ...  ART_AND_DESIGN    4.7    87510   8.7M  5,000,000+   Free    0  Everyone  Art & Design  August 1, 2018      1.2.4    4.0.3 and up
3             Sketch - Draw & Paint  ART_AND_DESIGN    4.5   215644   25M  50,000,000+   Free    0    Teen  Art & Design  June 8, 2018  Varies with device      4.2 and up
4  Pixel Draw - Number Art Coloring Book  ART_AND_DESIGN    4.3     967   2.8M  100,000+   Free    0  Everyone  Art & Design;Creativity  June 20, 2018      1.1    4.4 and up
```

Visualizziamo le proprietà del dataset:

```
In [301]: 1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
App           10841 non-null object
Category       10841 non-null object
Rating         9367 non-null float64
Reviews        10841 non-null object
Size           10841 non-null object
Installs       10841 non-null object
Type           10840 non-null object
Price          10841 non-null object
Content Rating 10840 non-null object
Genres          10841 non-null object
Last Updated   10841 non-null object
Current Ver    10833 non-null object
Android Ver    10838 non-null object
dtypes: float64(1), object(12)
memory usage: 1.1+ MB
```

Il dataset contiene 10841 osservazioni e 13 variabili. Osserviamo che molte delle variabili (tranne "Rating") sono di tipo "object", anche se rappresentano dei valori numerici (es., Reviews, Size, Installs e Price). Osservando le prime righe del DataFrame visualizzate sopra possiamo dedurre che:

- Size non è rappresentato come valore numerico in quanto contiene l'unità di misura "M";
- Installs è in realtà una variabile categorica (riporta un "+" alla fine, quindi indica la classe "più di xxxx installazioni");

Non ci sono motivazioni apparenti per cui Reviews e Rating non siano stati interpretati come numeri. Costruiamo una funzione filtro che identifichi se un valore è convertibile in un dato tipo o meno:

```
In [302]: 1 def cannot_convert(x, t=float):
2     try:
3         t(x)
4         return False
5     except:
6         return True
7 print(cannot_convert('12'))
8 print(cannot_convert('12f'))
```

```
False
True
```

Applichiamo il filtro alla colonna Review per visualizzare i valori che non possono essere convertiti:

```
In [303]: 1 list(filter(cannot_convert,data['Reviews']))
```

```
Out[303]: ['3.0M']
```

Possiamo sostituire questo valore con la sua versione per esteso mediante il metodo replace :

```
In [304]: 1 data['Reviews']=data['Reviews'].replace({'3.0M':3000000})
```

A questo punto possiamo convertire i tipi dei valori della colonna in interi:

```
In [305]: 1 data['Reviews']=data['Reviews'].astype(int)
```

Visualizziamo nuovamente le informazioni del DataFrame:

```
In [306]: 1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
App           10841 non-null object
Category       10841 non-null object
Rating         9367 non-null float64
Reviews        10841 non-null int64
Size           10841 non-null object
Installs       10841 non-null object
Type           10840 non-null object
Price          10841 non-null object
Content Rating 10840 non-null object
Genres          10841 non-null object
Last Updated   10841 non-null object
Current Ver    10833 non-null object
Android Ver    10838 non-null object
dtypes: float64(1), int64(1), object(11)
memory usage: 1.1+ MB
```

Reviews è adesso un intero. Eseguiamo una analisi simile sulla variabile Price:

```
In [307]: 1 list(filter(cannot_convert, data['Price']))[:10] #visualizziamo solo i primi 10 elementi
```

```
Out[307]: ['$4.99',
'$4.99',
'$4.99',
'$4.99',
'$3.99',
'$3.99',
'$6.99',
'$1.49',
'$2.99',
'$3.99']
```

Possiamo convertire le stringhe in float eliminando il dollaro iniziale mediante apply :

```
In [308]: 1 def strip_dollar(x):
2     if x[0]=='$':
3         return x[1:]
4     else:
5         return x
6 data['Price']=data['Price'].apply(strip_dollar)
```

Vediamo se ci sono ancora valori che non possono essere convertiti:

```
In [309]: 1 list(filter(cannot_convert, data['Price']))
```

```
Out[309]: ['Everyone']
```

Dato che non sappiamo come interpretare il valore "Everyone", sostituiamolo con un NaN:

```
In [310]: 1 data['Price']=data['Price'].replace({'Everyone':np.nan})
```

Adesso possiamo procedere alla conversione:

```
In [311]: 1 data['Price']=data['Price'].astype(float)
2 data.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
App           10841 non-null object
Category       10841 non-null object
Rating         9367 non-null float64
Reviews        10841 non-null int64
Size           10841 non-null object
Installs       10841 non-null object
Type           10840 non-null object
Price          10840 non-null float64
Content Rating 10840 non-null object
Genres          10841 non-null object
Last Updated   10841 non-null object
Current Ver    10833 non-null object
Android Ver    10838 non-null object
dtypes: float64(2), int64(1), object(10)
memory usage: 1.1+ MB
```

Modifichiamo anche "Size" eliminando la "M" finale:

```
In [312]: 1 data['Size']=data['Size'].apply(lambda x : x[:-1])
```

Verifichiamo l'esistenza di valori non convertibili:

```
In [313]: 1 list(filter(lambda x: pd.isna(x), data['Size']))[:10]
```

```
Out[313]: ['Varies with devic',
'Varies with devic']
```

Sostituiamo questi valori con NaN:

```
In [314]: 1 data['Size']=data['Size'].replace({'Varies with devic':np.nan})
```

Verifichiamo nuovamente la presenza di valori che non possono essere convertiti:

```
In [315]: 1 list(filter(lambda x: pd.isna(x), data['Size']))
```

```
Out[315]: ['1,000']
```

Possiamo rimuovere la virgola usata per indicare le migliaia e convertire in float:

```
In [316]: 1 data['Size']=data['Size'].apply(lambda x: float(str(x).replace(',', '')))
```

```
In [317]: 1 data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10841 entries, 0 to 10840
Data columns (total 13 columns):
App           10841 non-null object
Category       10841 non-null object
Rating         9367 non-null float64
Reviews        10841 non-null int64
Size           9146 non-null float64
Installs       10841 non-null object
Type           10840 non-null object
Price          10840 non-null float64
Content Rating 10840 non-null object
Genres          10841 non-null object
Last Updated   10841 non-null object
Current Ver    10833 non-null object
Android Ver    10838 non-null object
dtypes: float64(3), int64(1), object(9)
memory usage: 1.1+ MB
```

Valutiamo se è il caso di trasformare anche "Installs" in un valore numerico. Vediamo quanti valori univoci di "Installs" ci sono nel dataset:

```
In [318]: 1 data['Installs'].nunique()
```

```
Out[318]: 22
```

Abbiamo solo 22 valori, che paragonati alle 10841 osservazioni, indicano che "Installs" è una variabile molto quantizzata, per cui probabilmente ha senso considerarla come categorica, piuttosto che come un valore numerico.

Il DataFrame contiene dei `NaN`. A seconda dei casi può avere senso tenerli o rimuoverli. Dato che i dati sono molti, possiamo rimuoverli con `dropna`:

```
In [319]: 1 data=data.dropna()
```

Possiamo quindi iniziare a esplorare i dati con gli strumenti visti. Visualizziamo ad esempio i valori medi delle variabili numeriche per categoria:

In [320]: 1 data.groupby('Category').mean()

Out[320]:

Category	Rating	Reviews	Size	Price
ART_AND_DESIGN	4.381034	1.874517e+04	12.939655	0.102931
AUTO_AND_VEHICLES	4.147619	1.575057e+04	24.728571	0.000000
BEAUTY	4.291892	5.020243e+03	15.513514	0.000000
BOOKS_AND_REFERENCE	4.320139	2.815291e+04	23.543750	0.145069
BUSINESS	4.119919	2.497765e+04	26.217480	0.249593
COMICS	4.130612	1.254822e+04	34.626531	0.000000
COMMUNICATION	4.102844	5.549962e+05	60.765403	0.197773
DATING	3.957803	2.254489e+04	18.312717	0.086590
EDUCATION	4.387273	6.435159e+04	30.588182	0.163273
ENTERTAINMENT	4.146667	1.621530e+05	21.853333	0.033222
EVENTS	4.478947	3.321605e+03	23.213158	0.000000
FAMILY	4.190347	1.801297e+05	37.459963	1.390074
FINANCE	4.112030	3.903023e+04	31.249624	9.172444
FOOD_AND_DRINK	4.097619	5.103793e+04	24.163095	0.059405
GAME	4.269507	1.386276e+06	46.827823	0.281704
HEALTH_AND_FITNESS	4.223767	4.519072e+04	29.658296	0.154350
HOUSE_AND_HOME	4.162500	2.935998e+04	17.505357	0.000000
LIBRARIES_AND_DEMO	4.204918	1.632359e+04	225.267213	0.000000
LIFESTYLE	4.093571	3.109027e+04	28.650714	6.976429
MAPS_AND_NAVIGATION	4.013684	3.858909e+04	26.282105	0.157474
MEDICAL	4.184259	4.392454e+03	49.643827	3.077901
NEWS_AND_MAGAZINES	4.143787	5.826802e+04	14.948521	0.023550
PARENTING	4.347727	1.999950e+04	21.579545	0.113409
PERSONALIZATION	4.323381	1.257211e+05	50.115827	0.427338
PHOTOGRAPHY	4.147034	3.260313e+05	18.970763	0.331992
PRODUCTIVITY	4.143830	1.854692e+05	38.899149	0.225362
SHOPPING	4.227933	2.624461e+05	33.397207	0.030615
SOCIAL	4.257062	1.830882e+05	24.317514	0.011186
SPORTS	4.204858	2.128411e+05	30.159109	0.324818
TOOLS	4.010742	1.663130e+05	47.929068	0.290047
TRAVEL_AND_LOCAL	4.043125	4.824554e+04	25.357500	0.165687
VIDEO_PLAYERS	4.025862	2.074172e+05	22.881897	0.008534
WEATHER	4.241176	7.639225e+04	24.805882	0.459608

Domanda 20



Visualizzare il numero medio di review per tipo (variabile Type).

Risposta 20



In [321]: 1 data=pd.read_csv('https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv')

In [322]: 1 data[data['Ticket']=='345779']

Out[322]:

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
81	82	1	3 Sheerlinck, Mr. Jan Baptist	male	29.0	0	0	345779	9.5	NaN	S

Esercizi

Esercizio 1



Definire la lista `[1,8,2,6,15,21,76,22,0,111,23,12,24]`, dunque:

- Stampare il primo numero della lista;
- Stampare l'ultimo numero della lista;
- Stampare la somma dei numeri con indici dispari (e.g., 1,3,5,...) nella lista;
- Stampare la lista ordinata in senso inverso;
- Stampare la media dei numeri contenuti nella lista.

Esercizio 2

Si definisca un dizionario `mesi` che mappi i nomi dei mesi nei loro corrispettivi numerici. Ad esempio, il risultato di:



```
print mesi['Gennaio']
```

deve essere

1

Esercizio 3



Si considerino le seguenti liste:

```
l1 = [1,2,3]
l2 = [4,5,6]
l3 = [5,2,6]
```

Si combinino un ciclo for, zip e enumerate per ottenere il seguente output:

```
0 -> 10
1 -> 9
2 -> 15
```

Esercizio 4



Si ripeta l'esercizio 2 utilizzando la comprensione di dizionari. A tale scopo, si definisca prima la lista `['Gennaio', 'Febbraio', 'Marzo', 'Aprile', 'Maggio', 'Giugno', 'Luglio', 'Agosto', 'Settembre', 'Ottobre', 'Novembre', 'Dicembre']`.

Si costruisca dunque il dizionario desiderato utilizzando la comprensione di dizionari e la funzione enum

Esercizio 5



Date le variabili:

```
obj='triangolo'
area=21.167822
```

stampare le stringhe:

- L'area del triangolo è 21.16
- 21.1678 è l'area del triangolo

Utilizzare la formattazione di stringhe per ottenere il risultato.



Esercizio 6 Scrivere una funzione che estratta il dominio da un indirizzo email. Ad esempio, se l'indirizzo è "furnari@dmi.unict.it", la funzione deve estrarre "dmi.unict.it".

Esercizio 7



Definire una matrice 3x4, poi:

- Stampare la prima riga della matrice;
- Stampare la seconda colonna della matrice;
- Sommare la prima e l'ultima colonna della matrice;
- Sommare gli elementi lungo la diagonale principale;
- Stampare il numero di elementi della matrice.

Esercizio 8



Si generi un array di 100 numeri compresi tra 2 e 4 e si calcoli la somma degli elementi i cui quadrati hanno un valore maggiore di 8.

Esercizio 9



Si generi la seguente matrice scrivendo la minore quantità di codice possibile:

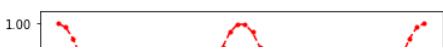
```
0 1 2
3 4 5
6 7 8
```

Si ripeta l'esercizio generando una matrice 25×13 dello stesso tipo.

Esercizio 10



Si scriva il codice per ottenere il seguente grafico:



Esercizio 11

Si carichi il dataset disponibile al seguente URL: <https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv> (<https://raw.githubusercontent.com/agconti/kaggle-titanic/master/data/train.csv>) in modo da utilizzare i valori della colonna "PassengerId" come indici e si applichino le seguenti trasformazioni:

- Si rimuovano le righe che contengono dei valori NaN;
- Si modifichi il DataFrame sostituendo nella colonna "Sex" ogni occorrenza di "male" con "M" e ogni occorrenza di "female" con "F";
- Si inserisca una nuova colonna "Old" con valore pari a "1" se Age è superiore all'età media dei passeggeri e "0" altrimenti;

• Si convertano i valori della colonna "Age" in interi.

Esercizio 12

Si consideri il dataset caricato e modificato nell'esercizio precedente. Si trovi il nome del passeggero con ticket dal codice "345779".



Esercizio 13

Quali sono i nomi dei 5 passeggeri più giovani?



Esercizio 14

Si consideri il dataset caricato e modificato nell'esercizio precedente. Si selezionino tutte le osservazioni relative a passeggeri di età compresa tra 20 e 30 anni.



Esercizio 15

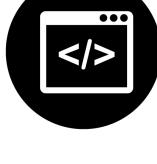
Si consideri il dataset caricato e modificato negli esercizi precedenti. Si calcoli la tariffa media pagata dai passeggeri di sesso femminile in seconda classe.



Esercizio 16

Si consideri il dataset caricato e modificato negli esercizi precedenti. Si costruisca una crosstab che permetta di rispondere alle seguenti domande:

- Quanti passeggeri di sesso maschile sono sopravvissuti?
- Quanti passeggeri di sesso femminile non sono sopravvissuti?



Esercizio 17

Si consideri il dataset caricato e modificato negli esercizi precedenti. Si costruisca una crosstab che permetta di rispondere alle seguenti domande:

- Quanti passeggeri di sesso maschile sono sopravvissuti in prima classe?
- Quanti passeggeri di sesso femminile sono sopravvissuti in terza classe?
- Quanti passeggeri di sesso maschile sono sopravvissuti in terza classe?



Esercizio 18

Si consideri il seguente DataFrame:

```
pd.DataFrame({'Category':[1,2,3], 'NumberOfElements':[3,2,3], 'CheckedElements':[1,2,1]})
```

Si costruisca un nuovo DataFrame che contenga le stesse colonne del DataFrame considerato e che per ogni riga di esso contenga `NumberOfElements` nuove righe con categoria uguale a `Category` di cui `CheckedElements` con valore di `CheckedElements` pari a uno e le restanti con valore di `CheckedElements` pari a zero.

Il risultato della manipolazione dovrà essere uguale al seguente dataset:

```
pd.DataFrame({'Category':[1,1,1,2,2,3,3,3],  
             'NumberOfElements':[1,1,1,1,1,1,1,1],  
             'CheckedElements':[1,0,0,1,1,0,0]})
```

Referenze

- Documentazione di Python 3: <https://docs.python.org/3/> (<https://docs.python.org/3/>)
- Documentazione di Numpy: <http://www.numpy.org/> (<http://www.numpy.org/>)
- Documentazione di Pandas: <https://pandas.pydata.org/pandas-docs/stable/> (<https://pandas.pydata.org/pandas-docs/stable/>)

