Machine Learning

Introduzione a Python per il Calcolo Scientifico e il Machine Learning

Antonino Furnari - http://www.dmi.unict.it/~furnari/ - furnari@dmi.unict.it
Giovanni Maria Farinella - http://www.dmi.unict.it/farinella/ - gfarinella@dmi.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 dimanico 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).

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 librerire che rendono la computazione numerica in Python comoda ed efficiente. Buona parte di queste librerie fanno parte di "SciPy" (https://www.scipy.org/), un ecosistema di software open-source per il calcolo scientifico. Altre librerie, quali ad esempio PyTorch (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.

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

- Python 3.7: https://docs.python.org/3.7/index.html;
- Numpy: http://www.numpy.org/;
- Matplotlib: https://matplotlib.org/;
- PyTorch: http://pytorch.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]: 3 + 5 #somma
```

```
Out[1]: 8
```

```
In [2]: 2 - 8 #differenza
```

```
Out[2]: -6
```

```
In [3]: 3 * 5 #prodotto
```

```
Out[3]: 15
```

```
In [4]: 3 / 2 #divisione, da notare che in Python 3, la divisione tra numeri interi restituisce un float
```

```
Out[4]: 1.5
```

```
In [5]: 3 // 2 #divisione intera
```

```
Out[5]: 1
```

```
In [6]: 9 ** 2 #elevamento a potenza
```

```
Out[6]: 81
```

In [7]:
```
4 % 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]:
```
x = 3
y = 9
z = 1.5
h = x/y
l = x//y
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]:
```
int(2.5)
```

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

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

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

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

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

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

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

In [13]:
```
a=1
a+=1
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]:
```
print(True)
print(False)
```

```
True
False
```

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

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

E' possibile generare booleani mediante gli operatori di confronto:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
True
True
False
```

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

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

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

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

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

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

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

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

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

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

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

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

1.4 Stampa

La stampa avviene mediante la funzione `print` :

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

```
2.2
```

Possiamo stampare una riga vuota omettendo il parametro di print:

In [30]:
```
print(2)
print()
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]:
```
print(2, end="\n\n")
print(3)
```

```
2

3
```

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

In [32]:
```
print(2, end="") #"" rappresenta una stringa vuota
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]:
```
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]:
```
l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
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]:
```
print(l[0],l[2])
l[0]=8 #assegnamento di un nuovo valore alla prima locazione di memoria
print(l)
```

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

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

In [36]:
```
l = []
print(l)
l.append(1)
l.append(2.5)
l.append(8)
l.append(-12)
print(l)
```

```
[]
[1, 2.5, 8, -12]
```

Le liste possono essere concatenate mediante l'operatore somma:

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

```
[1, 5, 4, 6]
```

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

In [38]:
```
l1 = [1,3]
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]:
```
print([0]*5) #lista di 5 zeri
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]:
```
print(l2)
print(len(l2))
```

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

Sulle liste è definito un ordinamento che dipende dalla loro lunghezza: liste più corte sono "minori di" liste più lunghe:

In [41]:
```
print([1,2,3]<[1,2,3,4])
print([1,2,3,5]>=[1,2,3,4])
```

```
True
True
```

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

In [42]:
```
print([1,2,3]==[1,2,3])
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]:
```
print(7 in [1,3,4])
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]:
```
l=[-5,2,10,6]
print(max(l))
print(min(l))
```

```
10
-5
```

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

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

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

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

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

In [47]:
```
l=[1,2,3,4,5]
print(l)
print(l.pop())
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]:
```
l = [1,2,3,4,5,6,7,8]
print("Lista l     ->", l)
print("l[0:3]      ->", l[0:3]) #dall'indice 0 (incluso) all'indice 3 (escluso)
print("l[1:2]      ->", l[1:2]) #dall'indice 1 (incluso) all'indice 2 (escluso)
```

```
Lista l     -> [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]:
```
print("l[:2]       ->", l[:2]) #dall'indice 0 (incluso) all'indice 2 (escluso)
#equivalente al seguente:
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]:
```
print("Ultimo indice della lista:",len(l))
print("l[3:]       ->", l[3:]) #dall'indice 3 (incluso) all'indice 5 (escluso)
#equivalente al seguente:
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]:
```
print("l[:]          ->", l[:]) #dall'indice 0 (incluso) all'indice 5 (escluso)
#equivalente a:
print("l[0:8]        ->", l[0:8]) #dall'indice 0 (incluso) all'indice 5 (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]:
```
print("l[0:8:2]    ->", l[0:8:2])
#da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento sì e uno no)
#equivalente a:
print("l[::2]      ->", l[0:8:2])
#da 0 (incluso) a 8 (escluso) a un passo di 2 (un elemento sì 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]:
```
print("l[5:2:-1]    ->", l[5:2:-1])
#da 5 (incluso) a 2 (escluso) a un passo di -1
print("l[2:5:-1]    ->", l[2:5:-1])
#in questo caso, il primo indice è più piccolo del secondo,
#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]:
```
print("l[:2:-1]     ->", l[:2:-1])
#dall'ultimo indice (incluso) a 2 (escluso) a un passo di -1
#equivalente a:
print("l[8:2:-1]    ->", l[8:2:-1])

print()
print("l[3::-1]     ->", l[3::-1])
#dal terzo indice (incluso) a 0 (incluso, in quanto omesso) a un passo di -1
#simile, ma non equivalente a:
print("l[3:0:-1]    ->", l[3:0:-1])
#dal terzo indice (incluso) a 0 (escluso) a un passo di -1

print()
print("l[::-1]      ->", l[::-1])
#dall'ultimo indice (incluso) al primo (incluso, in quanto omesso) a un passo di -1
#simile, ma non equivalente a:
print("l[8:0:-1]    ->", l[8:0:-1])
#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]:
```
print(l)
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]:
```
l = [5,7,9,-1,2,6,5,4,-6]
print(l)
l[3]=80
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]:
```
l[::2]=[0,0,0,0,0] #assegno 0 ai numeri di posizione dispari
print(l)
```

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

Le liste possono anche essere annidate:

In [58]:
```
a1 = [1,2,[4,8,[7,5]],[9],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]:
```
print(a1[2][2][0]) #il primo indice seleziona la lista [4,8,...]
#il secondo indice seleziona la lista [7,5]
#il terzo indice seleziona l'elemento 7
```

```
7
```

Domanda 2

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

```
l = [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]:
```
l = [1,2,3,4,5] #questa è una lista (parentesi quadre)
t = (1,2,3,4,5) #questa è una tupla (parentesi tonde)
print(l)
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]:
```
t = (1,3,5)
try:
    t[0]=8 #restituisce un errore in quanto le tuple sono immutabili
except Exception as e:
    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]:
```
t=(1)
print(t)
```

```
1
```

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

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

```
(1,)
```

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

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

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

E' possibile convertire tuple in liste e viceversa:

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

print(ttl)
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]:
```
t1=1,2,3

print(t1)

a,b,c=t1 #spacchettamento della tupla
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]:
```
var1 = "Var 1"
var2 = "Var 2"

print(var1,var2)

var1,var2=var2,var1
print(var1,var2)

#equivalente a:
var1 = "Var 1"
var2 = "Var 2"
t = (var2,var1)
var1=t[0]
var2=t[1]
print(var1,var2)
```

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

Le tuple annidate possono essere spacchetate come segue:

In [68]:
```
t = (1,(2,3),(4,5,6))
x,(t11,t12),(t21,t22,t23) = t
print(t)
print(x, t11, t12, t21, t22, t23)
#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]:
```
s1 = 'Singoli apici'
s2 = "Doppi apici, possono contenere anche apici singoli '' "
s3 = """Tripli
doppi apici
possono essere definite su più righe"""
type(s1), type(s2), type(s3)
```

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

La stampa avviene mediante la funzione "print":

In [70]:
```
print(s1)
print(s2)
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]:
```
print("ciao".upper()) #rendi tutto maiuscolo
print("CIAO".lower()) #tutto minuscolo
print("ciao come stai".capitalize()) #prima lettera maiuscola
print("ciao come stai".split()) #spezza una stringa e restituisce una lista
print("ciao, come stai".split(','))# spezza quando trova la virgola
print("-".join(["uno","due","tre"])) #costruisce una stringa concatenando gli elementi
#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]:
```
s = "Hello World"
print(s[:4]) #primi 4 caratteri
print(s[4:]) #dal quarto carattere alla fine
print(s[4:7]) #dal quarto al sesto carattere
print(s[::-1]) #inversione della stringa
```

```
Hell
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 `2018` dalla stringa `A-2017-B2` :

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

```
2017
```

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

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

```
True
False
```

Gli altri operatori rispecchiano l'ordinamento lessicografico tra stringhe:

In [75]:
```
print("abc"<"def")
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]:
```
#per costruire la stringa formattata, faccio seguire la stringa dal simbolo "%" e poi inserisco
#una tupla contenente gli argomenti
s1 = "Questa %s è formattata. Posso inserire numeri, as esempio %0.2f" % ("stringa",3.00002)
print(s1)
```

```
Questa stringa è formattata. Posso inserire numeri, as esempio 3.00
```

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

In [77]:
```
s2 = "Questa {} è formattata. Posso inserire numeri, ad esempio {}"\
    .format("stringa",3.000002)# il carattere "\" permette di spezzare la riga
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]:
```
print("Questa {:s} è formattata. Posso inserire numeri, ad esempio {:0.2f}"\
      .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 richimarli in maniera non ordinata:

In [79]:
```
print("Questa {str:s} è formattata. Posso inserire numeri, ad esempio {num:0.2f}"\
    .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

1.8 Dizionari

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

In [80]:
```
d = {"val1":1, "val2":2}
print(d)

print(d["val1"])
```

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

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

In [81]:
```
print(d.keys()) #chiavi e valori sono in ordine casuale
print(d.values())
```

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

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

In [82]:
```
d = {(2,3):5, (4,6):11}

print(d[(2,3)])
```

```
5
```

I dizionari possono anche essere estesi dinamicamente:

In [83]:
```
d = dict() #dizionario vuoto
d["chiave"]="valore"
print(d)
```

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

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

In [84]:
```
d = {1:'ciao', '5': 5, 8: -1}
print(5 in d)
print('5' in d)
```

```
False
True
```

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

In [85]:
```
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 [86]:
```
s = {1,2,3,3}
print(s) #può essere contenuto solo un "3"
```

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

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

In [87]:
```
print(s)
s.add(5)
print(s)
s.add(1) #non ha effetto. 1 è già presente
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 [88]:
```
s={1,5,-1}
print(-1 in s)
print(8 in s)
```

```
True
False
```

E' inoltre possibile creare set da liste:

In [89]:
```
set([1,3,3,2,5,1])
```

Out[89]:
```
{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 [90]:
```
var1 = 5
var2 = 10
if var1<var2:
    var3 = var1+var2
    print("Hello World",var3)
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 deliminare 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 [91]:
```
true_condition = True
false_condition = False

if true_condition: #i due punti ":" sono obbligatori
    word="cool!"
    print(word) #l'indentazione è obbligatoria

if false_condition:
    print("not cool :(")

if not false_condition: #neghiamo la condizione con "not"
    word="cool"
    print(word,"again :)")

if false_condition:
    word="this"
    print(word+" is "+"false")
else: #due punti + indentazione
    print("true")

if false_condition:
    print("false")
elif 5>4: #implementa un "else if"
    print("5>4")
else:
    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 [92]:
```
allowed_parameters = ["single", "double", 5, -2]

parameter = "double"

if parameter in allowed_parameters:
    print(parameter,"is ok")
else:
    print(parameter,"not in",allowed_parameters)

x=8
if x in allowed_parameters:
    print(x,"is not ok")
else:
    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 [93]:
```
var1 = 5
var2 = 3
m = var1 if var1>var2 else var2 #calcola il massimo
print(m)
```

```
5
```

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

In [94]:
```
s = "ciao"
if s=="help":
    print("help")
elif s=="world":
    print("hello",s)
elif s=="ciao":
    print(s,"mondo")
else:
    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 evidenziino 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 [95]:
```
i=0
while i<5: #due punti ":"
    print(i) #indentazione
    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 [96]:
```
l=[1,7,2,5]
for v in l:
    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 [97]:
```
for i in range(5):
    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 [98]:
```
print(range(5)) #oggetto di tipo range, genera numeri da 0 a 5 (escluso)
print(list(range(5))) #range non fa altro che generare numeri consecutivi
#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 [99]:
```
array=[1,7,2,4,5]
for i in range(len(array)):
    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 [100]:
```
for index,value in enumerate(array): #get both index and value
    print(index,"->",value)
```

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

Supponiamo adesso di voler scorrere contemporaneamente tutti gli iesimi elementi di più liste. Ad esempio:

In [101]:
```
a1=[1,6,2,5]
a2=[1,8,2,7]
a3=[9,2,5,2]

for i in range(len(a1)):
    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 [102]:
```
l1 = [1,6,5,2]
l2 = [3,8,9,2]
zipped=list(zip(l1,l2))
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 [103]:
```
for v1,v2,v3 in zip(a1,a2,a3):
    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 [104]:
```
for i,(v1,v2,v3) in enumerate(zip(a1,a2,a3)):
    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 [105]:
```
print(zip(l1,l2))
print(list(zip(l1,l2)))
```

```
<zip object at 0x7fc2b4049908>
[(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 [106]:
```
a = list(range(8))

b = [x*3 for x in a] #la lista "a" viene iterata. La variabile "x" conterrà di volta in volta i valori di
 a

print(a)
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 [107]:
```
print([x for x in a if x%2==0]) #include solo i numeri pari
```

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

In [108]:
```
a = [1,3,8,2,9]
b = [4,9,2,1,4]

c = [x+y for x,y in zip(a,b)]

print(a)
print(b)
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 [109]:
```
a = ["one","two","three","four","five","six","seven"]
b = range(1,8)
d = {i:s for i,s in zip(a,b)}
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 [110]:
```
def fun(x, y):
    return x**y

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 [111]:
```
def fun(x, y=2):
    return x**y

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 [112]:
```
print(fun(y=3,x=2))
```

```
8
```

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

In [113]:
```
def soMuchFun(x,y):
    return x**y, y**x

print(soMuchFun(2,3))

a,b=soMuchFun(2,3) #posso "spacchettare" la tupla restituita
print(a,b)
```

```
(8, 9)
8 9
```

E' inoltre possibile definire funzioni anonime come segue:

In [114]:
```
myfun = lambda x: x**2 #un input e un output
print(myfun(2))

myfun1 = lambda x,y: x+y #due input e un output
print(myfun1(2,3))

myfun2 = lambda x,y: (x**2,y**2) #due input e due output
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 [115]:
```
def pow2(x):
    return(x**2)

l1 = list(range(6))
l2 = list(map(pow2,l1)) #applica pow2 a tutti gli elementi della lista
print(l1)
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 [116]:
```
l2 = list(map(lambda x: x**2, l1))
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 [117]:
```
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 [118]:
```
class Classe(object): #ereditiamo dalla classe standard "object"
    def __init__(self, x): #costruttore
        self.x=x #inserisco il valore x nello stato dell'oggetto

    def prt(self):
        print(self.x)

    def power(self,y=2):
        self.x =  self.x**y

c = Classe(3)
c.prt()
c.power(3)
c.prt()
c.power()
c.prt()
```

```
3
27
729
```

Per estendere una classe, si fa come segue:

In [119]:
```
class Classe2(Classe): #derivo la classe "Classe" ed eredito metodi e proprietà
    def __init__(self,x):
        super(Classe2, self).__init__(x) #chiamo il costruttore della classe madre
    def prt(self): #ridefinisco il metodo prt
        print("yeah",self.x)

c2 = Classe2(2)
c2.power()
c2.prt()
```

```
yeah 4
```

1.16 Duck Typing

Per identificare i tipi dei dati, Python segue il principio del 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 [120]:
```
class Sparrow(object):
    def fly(self):
        print("Sparrow flying")

class Airplane(object):
    def fly(self):
        print("Airplane flying")

class Whale(object):
    def swim(self):
        print("Whale swimming")

def lift_off(entity):
    entity.fly()

sparrow = Sparrow()
airplane = Airplane()
whale = Whale()

try:
    lift_off(sparrow)
    lift_off(airplane)
    lift_off(whale) #Errore, il "tipo" di questo oggetto non permette di eseguire il metodo "fly".
                    #Secondo il duck test, questo tipo è incompatibile
except AttributeError as e:
    print("Error:",e)
```

```
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 [121]:
```
try:
    5/0
except:
    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 [122]:
```
try:
    5/0
except ZeroDivisionError:
    print("Houston, abbiamo un problema!")
```

```
Houston, abbiamo un problema!
```

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

In [123]:
```
try:
    5/0
except ZeroDivisionError as e:
    print("Houston, abbiamo un problema!")
    print(e)
```

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

E' possibile lanciare una eccezione mediante `raise` :

In [124]:
```
def div(x,y):
    if y==0:
        raise ZeroDivisionError("Cannot divide by zero!")
    else:
        return x/y
try:
    div(5,2)
    div(5,0)
except Exception as e:
    print(e)
```

```
Cannot divide by zero!
```

Possiamo definire nuove eccezioni estendendo la classe `Exception` :

In [125]:
```
class MyException(Exception):
    def __init__(self, message):
        self.message = message

try:
    raise MyException("Exception!")
except Exception as e:
    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 [126]:
```
def div(x,y):
    assert y!=0, "Cannot divide by zero!"
    return x/y

try:
    div(5,2)
    div(5,0)
except:
    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.

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 [127]:
```
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 [128]:
```
l = [[1,2,3],[4,5,2],[1,8,3]] #una lista contenente tre liste
print("List of lists:",l) #viene visualizzata così come l'abbiamo definita
a = np.array(l) #costruisco un array di numpy a partire dalla lista di liste
print("Numpy array:\n",a) #ogni lista interna viene identificata come una riga di una matrice bidimensiona
le
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 [129]:
```
print(a.shape) #si tratta di una matrice 3 x 3
```

```
(3, 3)
```

Vediamo qualche altro esempio

In [130]:
```
array = np.array([1,2,3,4])
matrice = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
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
```

```
Array: [1 2 3 4] (4,)
Matrix:
 [[1 2 3 4]
 [5 4 2 3]
 [7 5 3 2]
 [0 2 3 1]] (4, 4)
matrix:
 [[['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']]] (4, 2, 4)
```

Vediamo qualche operazione tra numpy array:

In [131]:
```
a1 = np.array([1,2,3,4])
a2 = np.array([4,3,8,1])
print("Sum:",a1+a2) #somma tra vettori
print("Elementwise multiplication:",a1*a2) #moltiplicazione tra elementi corrispondenti
print("Power of two:",a1**2) #quadrato degli elementi
print("Elementwise power:",a1**a2) #elevamento a potenza elemento per elemento
print("Vector product:",a1.dot(a2)) #prodotto vettoriale
print("Minimum:",a1.min()) #minimo dell'array
print("Maximum:",a1.max()) #massimo dell'array
print("Sum:",a2.sum()) #somma di tutti i valori dell'array
print("Product:",a2.prod()) #prodotto di tutti i valori dell'array
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 [132]:
```
m1 = np.array([[1,2,3,4],[5,4,2,3],[7,5,3,2],[0,2,3,1]])
m2 = np.array([[8,2,1,4],[0,4,6,1],[4,4,2,0],[0,1,8,6]])

print("Sum:",m1+m2) #somma tra matrici
print("Elementwise product:\n",m1*m2) #prodotto elemento per elemento
print("Power of two:\n",m1**2) #quadrato degli elementi
print("Elementwise power:\n",m1**m2) #elevamento a potenza elemento per elemento
print("Matrix multiplication:\n",m1.dot(m2)) #prodotto matriciale
print("Minimum:",m1.min()) #minimo
print("Maximum:",m1.max()) #massimo
print("Minimum along columns:",m1.min(0)) #minimo per colonne
print("Minimum along rows:",m1.min(1)) #minimo per righe
print("Sum:",m1.sum()) #somma dei valori
print("Mean:",m1.mean()) #valore medio
print("Diagonal:",m1.diagonal()) #diagonale principale della matrice
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 [133]:
```
a=np.linspace(10,20,5) # genera 5 valori equispaziati che vanno da 10 a 20
print(a)
```

```
[10.  12.5 15.  17.5 20. ]
```

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

In [134]:
```
print(np.arange(10)) #numeri da 0 a 9
print(np.arange(1,6)) #numeri da 1 a 5
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 [135]:
```
print(np.zeros((3,4)))#zeros e ones prendono come parametro una tupla contenente le dimensioni desiderate
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 [136]:
```
print(np.eye(3))
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 [137]:
```
print(np.random.rand(5)) #un array con 5 valori casuali tra 0 e 1
print(np.random.rand(3,2)) #una matrice 3x2 di valori casuali tra 0 e 1
```

```
[0.04940046 0.36590481 0.02036412 0.78869353 0.44060789]
[[0.17506439 0.5868475 ]
 [0.41717659 0.84880194]
 [0.61327861 0.60715505]]
```

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

In [138]:
```
print(np.random.randn(5,2))
```

```
[[-0.49148359 -0.75496029]
 [ 2.32634058  0.33077592]
 [ 0.38333046  0.35814408]
 [ 0.017981    1.03952501]
 [-2.2090141  -0.10050881]]
```

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

In [139]:
```
print(np.random.randint(0,50,3))#tre valori compresi tra 0 e 50 (escluso)
print(np.random.randint(0,50,(2,3)))#matrice 2x3 di valori interi casuali tra 0 e 50 (escluso)
```

```
[33 19 42]
[[28 44 19]
 [38 41 20]]
```

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

In [140]:
```
np.random.seed(123)
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 [141]:
```
np.random.seed(123)
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 [142]:
```
mat = np.array([[1,-5,0],[4,3,6],[5,8,-7],[10,6,-12]])
print(mat)
print()
print( mat.max(0))# massimi per colonne
print()
print(mat.max(1))# massimi per righe
print()
print (mat.min(0))# minimi per colonne
print()
print(mat.min(1))# minimi per righe
print()
print(mat.sum(0))# somma per colonne
print()
print(mat.sum(1))# somma per righe
print()
print(mat.max())# massimo globale
print(mat.min())# minimo globale
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 [143]:
```
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 [144]:
```
print(mat[3,0])
print(mat[2,1])
print(mat[1,2])
print(mat.max(0))
```

```
10
8
6
[10  8  6]
```

Analogamente:

In [145]:
```
print(mat.argmax(1))
print(mat.argmin(0))
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 [146]:
```
arr = np.array([1,2,3,4,5])
print("arr[0]       ->",arr[0]) #primo elemento dell'array
print("arr[:3]      ->",arr[:3]) #primi tre elementi
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 indicizza un array a più di una dimensioni con un unico indice, viene in automatico indicizzata la prima dimensione. Vediamo qualche esempio con le matrici bidimensionali:

In [147]:
```
mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
print("Matrice:\n",mat,mat.shape) #matrice 3 x 4
print("mat[0]     ->",mat[0]) #una matrice è una collezione di righe, per cui mat[0] restituisce la prima
 riga
print("mat[-1]     ->",mat[-1]) #ultima riga
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 [148]:
```
tens = np.array(([[1,5,2,7],
                  [2,7,3,2],
                  [1,5,2,1]],
                  [[1,5,2,7],
                  [2,7,3,2],
                  [1,5,2,1]]))
print("Matrice:\n",tens,tens.shape) #tensore 2x3x4
print("tens[0]    ->",tens[0])#si tratta della prima matrice 3x4
print("tens[-1]    ->",tens[-1])#ultima mtraice 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 [149]:
```
mat = np.array(([1,5,2,7],[2,7,3,2],[1,5,2,1]))
print("Matrice:\n",mat,mat.shape) #matrice 3 x 4
print("mat[2][1]  ->",mat[2][1]) #terza riga, seconda colonna
print("mat[0,0]   ->",mat[0,0]) #prima riga, prima colonna (notazione più compatta)

print("mat[0]     -> ",mat[0]) #restituisce l'intera prima riga della matrice
print("mat[:,0]   -> ",mat[:,0]) #restituisce la prima colonna della matrice.
                                 #I due punti ":" significano "lascia tutto inalterato lungo questa dimensi
one"
print("mat[0,:]   ->",mat[0,:]) #notazione alternativa per ottenere la prima riga della matrice
print("mat[0:2,:] ->\n",mat[0:2,:]) #prime due righe
print("mat[:,0:2] ->\n",mat[:,0:2]) #prime due colonne
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 [150]:
```
mat=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("mat[:,:,0] ->", mat[:,:,0]) #matrice 3 x 3 contenuta nel "primo canale" del tensore
print("mat[:,:,1] ->",mat[:,:,1]) #matrice 3 x 3 contenuta nel "secondo canale" del tensore
print("mat[...,0] ->", mat[...,0]) #matrice 3 x 3 contenuta nel "primo canale" del tensore (notazione alte
rnativa)
print("mat[...,1] ->",mat[...,1]) #matrice 3 x 3 contenuta nel "secondo canale" del tensore (notazione alt
ernativa)
#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 estrae 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 [151]:
```
x = np.array([1,2,3])
print(x[np.array([True,False,True])]) #per selezionare solo 1 e 3
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 [152]:
```
x = np.arange(10)
print(x)
print(x>2) #genera un array di valori booleani
#che conterrà True in presenza dei valori di x
#che verificano la condizione x>2
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  True]
[False False False  True False False False False False False]
```

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

In [153]:
```
x = np.arange(10)
print(x[x%2==0]) #seleziona i valori pari
print(x[x%2!=0]) #seleziona i valori dispari
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 [154]:
```
mat = np.array([[1,2],[3,4],[5,6]])
print(mat)
print(mat.reshape(2,3))
print(mat.reshape(1,6))
print(mat.reshape(6,1)) #matrice 6 x 1
print(mat.reshape(6)) #vettore unidimensionale
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 lascire che numpy clacoli una delle dimensioni sostituendola con -1:

In [155]:
```
print(mat.reshape(2,-1))
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 [156]:
```
mat1 = np.random.rand(3,2)
mat2 = np.random.rand(2,3)
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 [157]:
```
a=np.arange(9).reshape(3,3)
print(a,a.shape,"\n")
cat=np.concatenate([a,a])
print(cat,cat.shape,"\n")
cat2=np.concatenate([a,a,a])
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 [158]:
```
a=np.arange(9).reshape(3,3)
print(a,a.shape,"\n")
cat=np.concatenate([a,a], axis=1) #concatenazione per colonne
print(cat,cat.shape,"\n")
cat2=np.concatenate([a,a,a], axis=1) #concatenazione per colonne
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 [159]:
```
print(cat.shape,a.shape) #concatenazione lungo l'asse 0, le dimensioni lungo gli altri assi devono essere
 uguali
```

```
(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 [160]:
```
cat=np.concatenate([a,a])
print(cat,cat.shape)
stack=np.stack([a,a])
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 [161]:
```
stack=np.stack([a,a],axis=1)
print(stack,stack.shape)
```

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

 [[3 4 5]
  [3 4 5]]

 [[6 7 8]
  [6 7 8]]] (3, 2, 3)
```

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

In [162]:
```
stack=np.stack([a,a],axis=2)
print(stack,stack.shape)
```

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

 [[3 3]
  [4 4]
  [5 5]]

 [[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 per la lista di tipi supportati). Possiamo vedere il tipo di un array ispezionando la proprietà dtype:

In [163]:
```
print(mat1.dtype)
```

```
float64
```

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

In [164]:
```
mat = np.array([[1,2,3],[4,5,6]],int)
print(mat.dtype)
```

```
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 [165]:
```
print(mat/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 [166]:
```
a=np.array([[1,2,3],[4,5,6]])
print(a)
b=a[0,0:2]
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 [167]:
```
b[0]=-1
print(b)
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 [168]:
```
a=np.array([[1,2,3],[4,5,6]])
print(a)
b=a[0,0:2].copy()

print(b)
b[0]=-1
print(b)
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 [169]:
```
mat=np.array([[1,2,3],[4,5,6]],dtype=np.float)
arr=np.array([2,3,8])
print(mat)
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 [170]:
```
mat2=mat.copy() #copia il contenuto della matrice per non sovrascriverla
for i in range(mat2.shape[0]):#indicizza le righe
    mat2[i]=mat2[i]/arr
print(mat2)
```

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

In [171]:
```
arr.shape
```

Out[171]:
```
(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 [172]:
```
arr2=np.stack([arr,arr])
print(arr2)

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 [173]:
```
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 [174]:
```
print(2*mat)
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:

$$\begin{matrix} 2 \times 3 \times 5 \\ 2 \times 3 \times 5 \end{matrix}$$

$$\begin{matrix} 2 \times 3 \times 5 \\ 2 \times 1 \times 5 \end{matrix}$$

$$\begin{matrix} 2 \times 3 \times 5 \\ \quad 3 \times 5 \end{matrix}$$

$$\begin{matrix} 2 \times 3 \times 5 \\ \quad 3 \times 1 \end{matrix}$$

Vediamo altri esempi di broadcasting:

In [175]:
```
mat1=np.array([[[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]]])
mat2=np.array([[2,1,3],[7,6,2]])
print("Mat1 shape",mat1.shape)
print("Mat2 shape",mat2.shape)
print()
print("Mat1\n",mat1)
print()
print("Mat2\n",mat2)
print()
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 [176]:
```
mat1=np.array([[[1,3,5],[7,6,2]],[[6,5,2],[8,9,9]]])
mat2=np.array([[[1,3,5]],[[6,5,2]]])
print("Mat1 shape",mat1.shape)
print("Mat2 shape",mat2.shape)
print()
print("Mat1\n",mat1)
print()
print("Mat2\n",mat2)
print()
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 coppi di valori (x, y) che verfichino l'equazione riportata sopra. Il modo più semplice per farlo consiste nel:

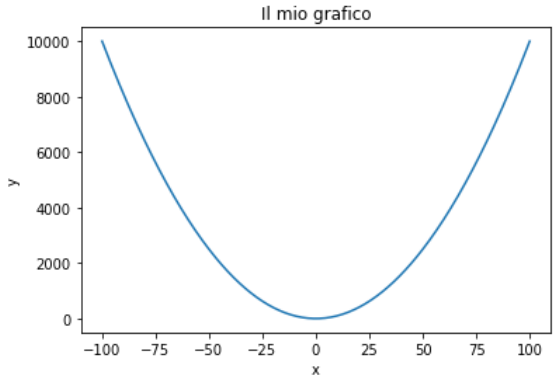
- Definire un vettore arbitrario di valori x estratti dal dominio della funzione;
- Calcolare i rispettivi punti y utilizzando la forma analitica riportata sopra;
- Eseguire il plot dei valori mediante matplotlib.

I due vettori possono essere definiti così:

In [233]:
```
x = np.linspace(-100,100,300) #campioniamo in maniera lineare 300 punti tra -100 e 100
y = x**2 #calcoliamo il quadrato di ognuno dei punti di x
```

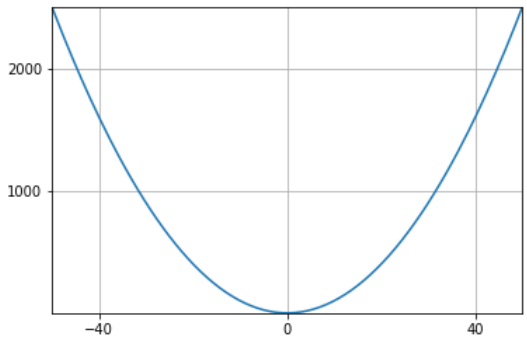
Procediamo alla stampa:

In [234]:
```
from matplotlib import pyplot as plt #importo il modulo pyplot da matplotlib e lo chiamo plt

plt.plot(x,y) #stampa le coppie x,y come punti nello spazio cartesiano
plt.xlabel('x') #imposta una etichetta per l'asse x
plt.ylabel('y') #imposta una etichetta per l'asse y
plt.title('Il mio grafico') #imposta il titolo del grafico
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 [179]:
```
plt.plot(x,y) #stampiamo lo stesso plot
plt.xlim([-50,50]) #visualizziamo i valori x solo tra -20 e 40
plt.ylim([0,2500]) #e i valori y solo tra 0 e 4000
plt.xticks([-40,0,40]) #inseriamo solo -40, 0 e 40 come "tick" sull'asse x
plt.yticks([1000,2000]) #solo 1000 e 2000 come tick y
plt.grid() #aggiungiamo una griglia
plt.show()
```



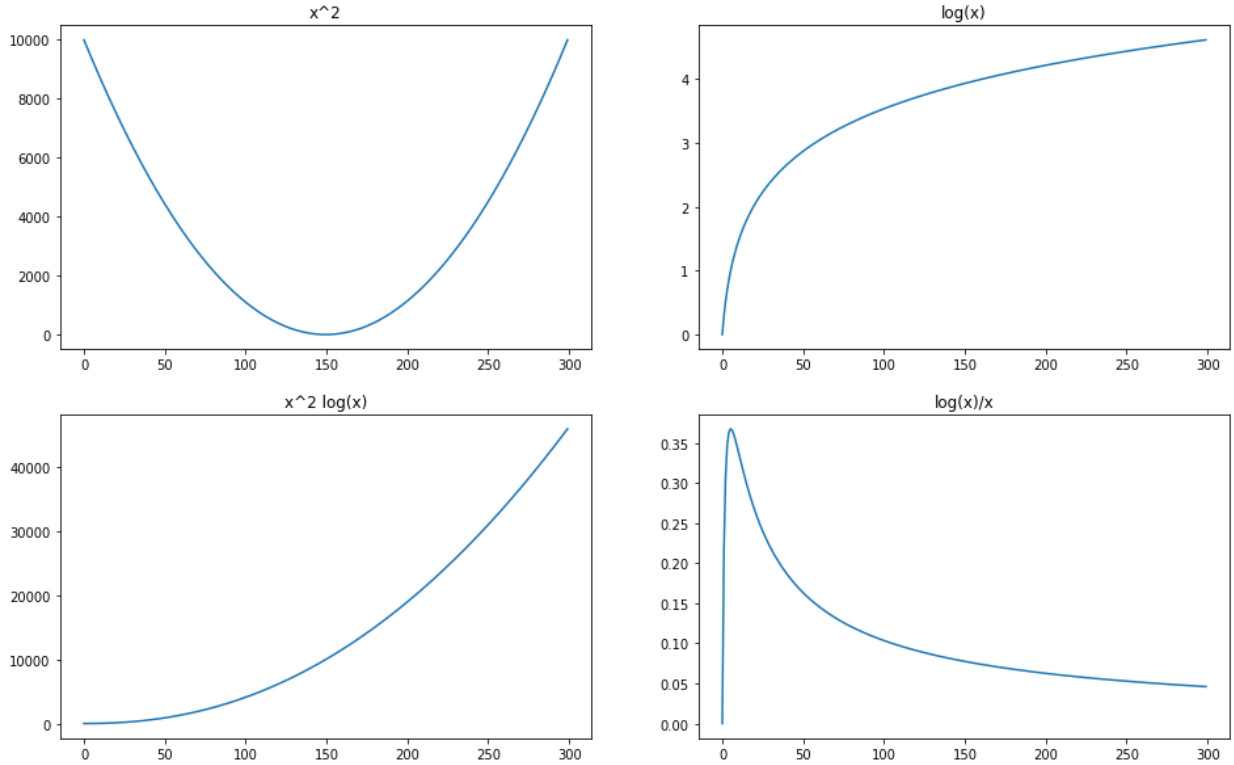
3.2 Subplot

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 x 2:

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

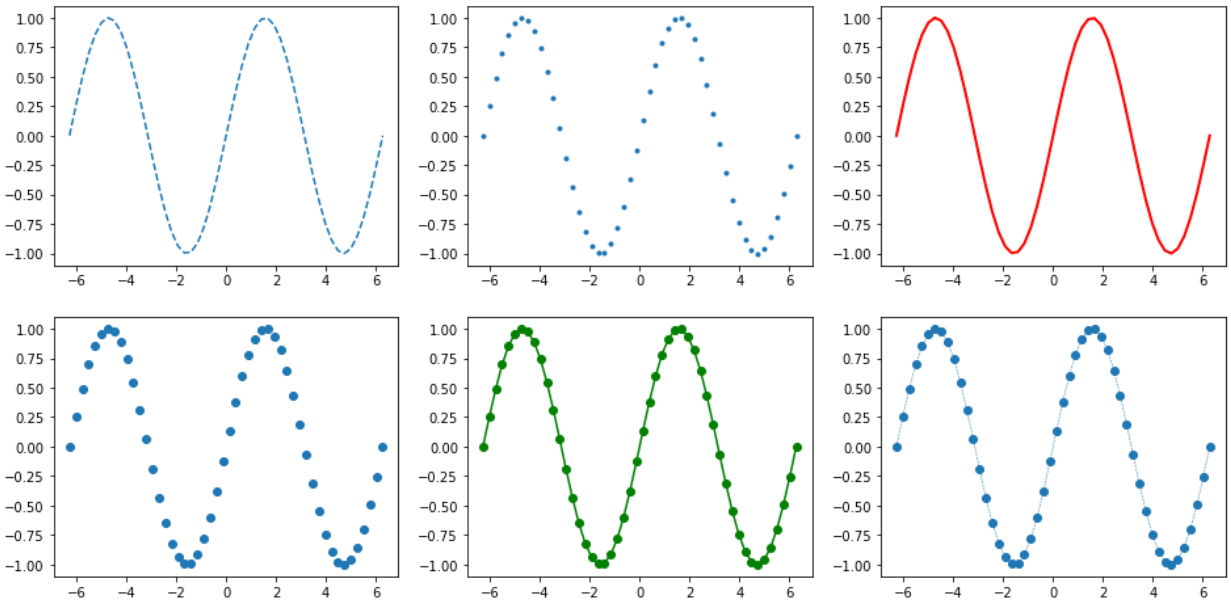
In [180]:
```
plt.figure(figsize=(16,10)) #definisce una figura con una data dimensione (in pollici)
plt.subplot(2,2,1) #definisce una griglia 2x2 e seleziona la prima cella come cella corrente
x = np.linspace(-100,100,300)
plt.plot(x**2) #primo grafico
plt.title('x^2') #imposta un titolo per la cella corrente

plt.subplot(2,2,2) #sempre nella stessa grliglia 2x2, seleziona la seconda cella
x = np.linspace(1,100,300)
plt.plot(np.log(x)) #secondo grafico
plt.title('log(x)') #imposta un titolo per la cella corrente

plt.subplot(2,2,3) #seleziona la terza cella
x = np.linspace(1,100,300)
plt.plot(x**2*np.log(x)) #terzo grafico
plt.title('x^2 log(x)') #imposta un titolo per la cella corrente

plt.subplot(2,2,4) #seleziona la quarta cella
x = np.linspace(1,100,300)
plt.plot(np.log(x)/x) #quarto grafico
plt.title('log(x)/x') #imposta un titolo per la cella corrente

plt.show() #mostra la figura
```



3.3 Colori e stili

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

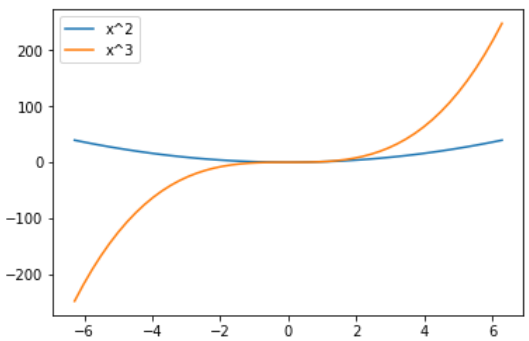
In [181]:
```
x = np.linspace(-2*np.pi,2*np.pi,50)
y = np.sin(x)

plt.figure(figsize=(16,8))
plt.subplot(231) #versione abbreviata di plt.subplot(2,3,1)
plt.plot(x,y,'--') #linea tratteggiata
plt.subplot(232)
plt.plot(x,y,'.') #solo punti
plt.subplot(233)
plt.plot(x,y,'r', linewidth=2) #linea rossa, spessore 2
plt.subplot(234)
plt.plot(x,y,'o') #cerchietti
plt.subplot(235)
plt.plot(x,y,'o-g') #cerchietti uniti da linee verdi
plt.subplot(236)
plt.plot(x,y,'o--', linewidth=0.5) #cerchietti uniti da linee tratteggiate, spessore 0.5

plt.show()
```



3.4 Sovrapposizione di linee e legenda

E' semplice sovrapporre più linee mediante **matplotlib**:

In [182]:
```
plt.figure()
plt.plot(x,x**2)
plt.plot(x,x**3) #al secondo plot, matplotlib cambierà automaticamente colore
plt.legend(['x^2','x^3']) #possiamo anche inserire una legenda.
#GLi elementi della legenda verranno associati alle linee nell'ordine in cui sono stati plottati
plt.show()
```



Matplotlib è una librerira 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).

4. Introduzione a PyTorch

PyTorch è una libreria Python per il calcolo scientifico pensata per due usi:

- In sostituzione a Numpy, per sfruttare tutte le potenzialità del calcolo parallelo su GPU;
- Come piattaforma per il deep learning.

4.1 Tensori

L'elemento base di PyTorch è il tensore. Si tratta di un tipo di dato molto simile al Numpy array, che ci permette di gestire tensori multidimensionali. Esempi di tensori sono gli array (tensori monodimensionali) e le matrici (tensori bidimensionali).

Per lavorare con i tensori in PyTorch, importiamo il modulo `torch`:

In [183]:
```
import torch
```

Possiamo costruire un tensore non inizializzato come segue:

In [184]:
```
x = torch.Tensor(2, 3)
print(x)
```

```
tensor([[-3.8565e-03,  4.5831e-41, -2.3891e+33],
        [ 4.5916e-41,  4.4842e-44,  0.0000e+00]])
```

Dato che il tensore non è inizializzato, il suo contenuto può essere qualsiasi (attenzione, non abbiamo la certezza che il contenuto sia neanche "casuale" - in generale esso dipende dallo stato attuale del sistema).

Possiamo costruire un tensore inizializzato in maniera casuale usando la funzione `rand`:

In [185]:
```
x = torch.rand(2, 3)
print(x)
```

```
tensor([[0.6185, 0.8912, 0.5438],
        [0.5136, 0.0390, 0.5465]])
```

Similmente a quanto possibile con Numpy, possiamo creare tensori di zeri e tensori di uno come segue:

In [186]:
```
print(torch.zeros(2,3))
print(torch.ones(2,3))
```

```
tensor([[0., 0., 0.],
        [0., 0., 0.]])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

Analogamente a quanto avviene per Numpy, è possibile creare tensori a partire da liste di Python:

In [187]:
```
torch.Tensor([[1,2,3],[4,5,6]])
```

Out[187]:
```
tensor([[1., 2., 3.],
        [4., 5., 6.]])
```

4.1.1 Tipi dei tensori

Così come in numpy, anche in PyTorch ogni tensore ha un tipo. In particolare sono disponibili i seguenti tipi:

FloatTensor	numeri in virgola mobile a 32 bit
DoubleTensor	numeri in virgola mobile a 64 bit
HalfTensor	numeri in virgola mobile a 16 bit
ByteTensor	numeri interi senza segno a 8 bit
CharTensor	numeri interi con segno a 8 bit
ShortTensor	numeri interi con segno a 16 bit
IntTensor	numeri interi con segno a 32 bit
LongTensor	numeri interi con segno a 64 bit

Maggiori informazioni sono disponibili qui: http://pytorch.org/docs/master/tensors.html#torch-tensor

Ogni tipo ha un costruttore dedicato, per cui se vogliamo costruire un tensore di Int possiamo scrivere:

```
In [188]: torch.IntTensor([1,2,3])
```

```
Out[188]: tensor([1, 2, 3], dtype=torch.int32)
```

In pratica `Tensor` è un alias a `FloatTensor` , per cui ogni tensore che abbiamo costruito negli esempi precedenti è di tipo `FloatTensor` . Possiamo ottenere il tipo di un tensore mediante il metodo `type()` :

```
In [189]: print(x.type())
```

```
torch.FloatTensor
```

4.1.2 Shape

Analogamente a quanto avviene con Numpy, ogni tensore ha una "forma" (shape). Questa si può richiamare mediante la proprietà `shape` o mediante il metodo `size` :

```
In [190]: print(x.shape)
          print(x.size())
```

```
torch.Size([2, 3])
torch.Size([2, 3])
```

Tutte le shape sono di tipo `tensor.Size` . Il metodo size, permette di accedere ai singoli elementi della shape passandone gli indici tra parentesi, per cui le seguenti notazioni sono equivalenti:

```
In [191]: print(x.shape[0], x.size(0))
          print(x.shape[1], x.size(1))
```

```
2 2
3 3
```

Come in Numpy, anche in PyTorch è possibile cambiare la shape di un tensore. In questo caso però non utilizziamo il metodo `reshape` , ma il metodo `view` . Tale metodo **non crea** una nuova copia del tensore, ma bensì fornisce **una nuova vista** del tensore (in pratica la posizione degli elementi del tensore in memoria resta invariata). Vediamo qualche esempio:

```
In [192]: print(x.view(-1))
          print(x.view(3,2))
          print(x.view(1,-1))
```

```
tensor([0.6185, 0.8912, 0.5438, 0.5136, 0.0390, 0.5465])
tensor([[0.6185, 0.8912],
        [0.5438, 0.5136],
        [0.0390, 0.5465]])
tensor([[0.6185, 0.8912, 0.5438, 0.5136, 0.0390, 0.5465]])
```

Domanda 9

Che dimensioni avranno i tensori costruiti come segue?

```
torch.Tensor([[1,2,3],[5,4,3]])
torch.Tensor([[[2,1],[3,3],[2,3]],[[2,2],[3,2],[4,5]]])
torch.Tensor([1,2,3])
```

Risposta 9

4.1.3 Operazioni tra tensori

Così come in Numpy, i tensori supportano delle operazioni, quali ad esempio la somma, la sottrazione, la moltiplicazione elemento per elemento (prodotto di Hadamard), la divisione elemento per elemento e la moltiplicazione riga per colonna. Vediamo qualche esempio:

In [193]:
```
x = torch.Tensor([[1,4,5],[4,2,8]])
y = torch.Tensor([[3,3,2],[2,1,8]])

print(x)
print(y)
print(x+y)
print(x-y)
print(x*y)
print(x/y)
```

```
tensor([[1., 4., 5.],
        [4., 2., 8.]])
tensor([[3., 3., 2.],
        [2., 1., 8.]])
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
tensor([[-2.,  1.,  3.],
        [ 2.,  1.,  0.]])
tensor([[ 3., 12., 10.],
        [ 8.,  2., 64.]])
tensor([[0.3333, 1.3333, 2.5000],
        [2.0000, 2.0000, 1.0000]])
```

Esistono altre due sintassi per effettuare le operazioni di somma. Una, equivalente a quella appena vista, è la seguente:

In [194]:
```
x = torch.Tensor([[1,4,5],[4,2,8]])
y = torch.Tensor([[3,3,2],[2,1,8]])


print(x)
print(y)
print(torch.add(x,y)) #x+y
print(torch.add(x,-y)) #x-y
print(torch.mul(x,y)) #x*y
print(torch.div(x,y)) #x/y
```

```
tensor([[1., 4., 5.],
        [4., 2., 8.]])
tensor([[3., 3., 2.],
        [2., 1., 8.]])
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
tensor([[-2.,  1.,  3.],
        [ 2.,  1.,  0.]])
tensor([[ 3., 12., 10.],
        [ 8.,  2., 64.]])
tensor([[0.3333, 1.3333, 2.5000],
        [2.0000, 2.0000, 1.0000]])
```

L'altra notazione è quella che ci permette di fare operazioni "in-place", ovvero, operazioni del tipo:

```
x+=y
```

Le operazioni in-place si effettuano utilizzando degli specific metodi dei tensori che terminano per "_". Il carattere underscore serve a distinguere le operazioni in-place da quelle non in-place:

In [195]:
```
print(x)
print(y)
x.add_(y)
print(x)
```

```
tensor([[1., 4., 5.],
        [4., 2., 8.]])
tensor([[3., 3., 2.],
        [2., 1., 8.]])
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
```

Notiamo come l'operazione in-place ha modificato il contenuto di "x". Altre operazioni in-place sono:

In [196]:
```
print(x)
x.mul_(y)
print(x)
x.div_(y)
print(x)
```

```
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
tensor([[ 12.,  21.,  14.],
        [ 12.,   3., 128.]])
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
```

Inserire l'underscore è importante, se lo omettiamo l'operazione non è più in-place:

In [197]:
```
print(x)
x.mul(y)
print(x) #x non ha cambiato valore
```

```
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
tensor([[ 4.,  7.,  7.],
        [ 6.,  3., 16.]])
```

Inoltre, i tensori di PyTorch supportano le comuni operazioni supportate da Numpy:

In [198]:
```
print(x.mean()) #media
print(x.mean(0)) #media per colonne
print(x.sum()) #somma
print(x.max()) #massimo
print(x.min()) #minimo
```

```
tensor(7.1667)
tensor([ 5.0000,  5.0000, 11.5000])
tensor(43.)
tensor(16.)
tensor(3.)
```

In PyTorch, la funzione `max` permette di effettuare contemporaneamente le operazioni `max` e `argmax`. In particolare, quando si passa un argomento a `max`, questa funzione restituisce due elementi: i valori massimi e gli indici in corrispondenza dei quali si hanno tali massimi:

In [199]:
```
x = torch.Tensor([[-5,1,0],[0,2,-2]])
valori, indici = x.max(0)
print(x)
print(valori)
print(indici)
```

```
tensor([[-5.,  1.,  0.],
        [ 0.,  2., -2.]])
tensor([0., 2., 0.])
tensor([1, 1, 0])
```

4.1.4 Indicizzazione

L'indicizzazione di tensori PyTorch avviene esattamente come l'indicizzazione di array Numpy. Alcuni esempi:

In [200]:
```
print(x)
print(x[0])
print(x[1,:])
print(x[1,1])
```

```
tensor([[-5.,  1.,  0.],
        [ 0.,  2., -2.]])
tensor([-5.,  1.,  0.])
tensor([ 0.,  2., -2.])
tensor(2.)
```

4.1.5 Numpy bridge

Abbiamo visto che tensori di PyTorch e array di Numpy sono simili in molti punti. In pratica può essere utile convertire un array da Numpy a PyTorch e viceversa. Ciò è possibile mediante degli specifici metodi. Possiamo convertire un tensore di PyTorch in un array di Numpy richiamando il metodo `numpy` :

In [201]:
```
print(x)
print(x.numpy() )
```

```
tensor([[-5.,  1.,  0.],
        [ 0.,  2., -2.]])
[[-5.  1.  0.]
 [ 0.  2. -2.]]
```

Analogamente, possiamo utilizzare la funzione `from_numpy` del modulo `torch` per convertire un array di numpy in un tensore di PyTorch:

In [202]:
```
import numpy as np
a = np.array([1,2,3])
b = torch.from_numpy(a)
print(a)
print(b)
```

```
[1 2 3]
tensor([1, 2, 3])
```

Esiste una corrispondenza tra i tipi di PyTorch e i tipi di Numpy. In particolare vediamo che l'array di int che abbiamo creato è stato automaticamente convertito in un LongTensor:

In [203]:
```
print(a.dtype)
print(b.type())
```

```
int64
torch.LongTensor
```

La corrispondenza completa tra tipi di Numpy e tipi di PyTorch è la seguente:

Tipo di Numpy	Tipo di PyTorch
int16	ShortTensor
int32	IntTensor
int64	LongTensor
uint8	ByteTensor
float32	FloatTensor
float64	DoubleTensor
double	DoubleTensor

Dopo la conversione, i tensori di PyTorch e gli array di Numpy corrispondenti condividono le stesse locazioni di memoria, per cui modificarne uno comporta modificarli entrambi. Vediamo un esempio:

In [204]:
```
a = np.array([[1,3,6],[7,7,7]],'int16')
b = torch.from_numpy(a)

print(a,b)

b[1,2]=-1
print(a,b)
```

```
[[1 3 6]
 [7 7 7]] tensor([[1, 3, 6],
        [7, 7, 7]], dtype=torch.int16)
[[ 1  3  6]
 [ 7  7 -1]] tensor([[ 1,  3,  6],
        [ 7,  7, -1]], dtype=torch.int16)
```

Come possiamo notare, entrambe le matrici sono state modificate. Lo stesso vale quando la conversione avviene in senso opposto:

In [205]:
```
a = torch.LongTensor([[1,2,3],[3,3,3]])
b = a.numpy()
print(a,b)

b[0,0]=-8
print(a,b)
```

```
tensor([[1, 2, 3],
        [3, 3, 3]]) [[1 2 3]
 [3 3 3]]
tensor([[-8,  2,  3],
        [ 3,  3,  3]]) [[-8  2  3]
 [ 3  3  3]]
```

4.1.6 Supporto GPU

Tutti i tensori vengono di default istanziati nella memoria di sistema. Le operazioni tra questi tensori vengono dunque eseguite in CPU. Se il supporto alla GPU è disponibile, è possibile spostare i tensori in GPU mediante il metodo `to(device)` , specificando come device `cuda` . Se entrambi i tensori sono in GPU, l'operazione viene effettuata interamente in GPU (e quindi sarà più veloce!). Possiamo controllare che il supporto alla GPU sia effettivamente abilitato mediante il metodo `torch.cuda.is_available()` . Vediamo un esempio:

In [206]:
```
x = torch.Tensor([1,2,3])
y = torch.Tensor([4,5,6])

if torch.cuda.is_available():
    x = x.to('cuda')
    y = y.to('cuda')
    print(x + y) #questa operazione viene effettuata in GPU
```

E' possibile spostare nuovamente i tensori in CPU specificando `cpu` come device:

In [207]:
```
x.to('cpu')
y.to('cpu')
print(x+y) #questa operazione viene effettuata in CPU
```

```
tensor([5., 7., 9.])
```

Domanda 10

Si consideri il seguente tensore:

`t = torch.Tensor([[4,8,2],[6,6,9]])`

- Qual è il tipo di t?
- Qual è il risultato dell'operazione `t[1,0]` ?
- Qual è il risultato dell'operazione `(t**2)[0]` ?
- Qual è la differenza tra le operazioni `t/2` e `t/=2` ? Quale delle due operazioni è più efficiente in termini di memoria?

Risposta 10

Domanda 11

Si consideri il seguente codice

```
t = torch.Tensor([[4,8,2],[6,6,9]])
n = t.numpy()
n/=2
```

Qual è il valore di `t[0,2]` ?

Risposta 11

4.2 Autograd

Autograd è il pacchetto di Python che permette la differenziazione automatica. Dal momento che le reti neurali sono assimilabili a funzioni composte, la differenziazione automatica ci risparmia la fatica di calcolare le derivate manualmente per ottimizzare i parametri mediante metodi basati sulla discesa del gradiente. Per permettere la differenziazione automatica, ogni tensore di PyTorch contiene le seguenti due proprietà:

- `grad` : il gradiente relativo al tensore. Per i tensori appena creati, questa proprietà sarà impostata a `None` in quanto nessun gradiente è stato ancora calcolato;
- `grad_fn` : un riferimento alla funzione che ha generato il tensore, nel caso in cui esso sia frutto di una operazione tra tensori (es. una somma). Se il tensore è stato generato dall'utente, allora `grad_fn` sarà `None` .

Il calcolo dei gradienti non è abilitato di default sui tensori. Per abilitalo, bisognerà richiamare il metodo `requires_grad_(True)` . Vediamo un esempio:

In [208]:
```
from torch.autograd import Variable
x = torch.Tensor([3])
x.requires_grad_(True)

print("Tensore:",x)
print("Gradiente:",x.grad)
print("Funzione che ha generato il tensore:",x.grad_fn)
```

```
Tensore: tensor([3.], requires_grad=True)
Gradiente: None
Funzione che ha generato il tensore: None
```

Inizialmente `grad_fn` è `None` perché il tensore è stato creato manualmente dall'utente. Anche `grad` è `None` perché ancora non abbiamo eseguito nessuna differenziazione. Il tensore appena creato è assimilabile a una **variabile indipendente**, che contiene un valore arbitrario. Vediamo di costruire una variabile **dipendente**, il cui valore dipende da quello di `x` . Ad esempio:

$$y = 2x^2$$

In [209]:
```
y =2*x**2
```

Ispezioniamo le proprietà del nuovo tensore `y` :

In [210]:
```
print(y)
print(y.grad_fn)
print(y.grad)
```

```
tensor([18.], grad_fn=<MulBackward0>)
<MulBackward0 object at 0x7fc28b568da0>
None
```

Il valore del tensore è 18 (ovvero il valore di x^2 moltiplicato per 2). `grad_fn` adesso punta a un oggetto chiamato `MulBackward0` . Ciò indica che la variabile `y` è stata creata moltiplicando una costante per una variabile. `grad` invece è ancora `None` . Per "riempire" il valore di `grad` , dobbiamo avviare il processo di differenziazione. Prima però riflettiamo su un punto: qual è la derivata di y rispetto a x?

$$y'(x) = 4x$$

La derivata di y rispetto a x è una funzione dipendente dalla variabile x. Qual è il valore della derivata di y rispetto a x calcolato nel punto $x = 3$?

$$y'(2) = 4 \cdot 3 = 12$$

Nel caso generale a più variabili, questo valore è detto gradiente. Avviamo il processo di differenziazione mediante il metodo `backward` . Questo metodo va richiamato sulla variabile `y` , in quanto

In [211]:
```
y.backward()
```

Il processo di differenziazione in pratica ha:

- Calcolato la derivata di y rispetto a tutte le variabili indipendenti;
- Valutato la funzione derivata nei valori assunti dalle variabili indipendenti.

Il gradiente rispetto alla variabile dipendente x si trova adesso dentro `x.grad` :

In [212]:
```
print(x.grad)
```

```
tensor([12.])
```

Come ci aspettiamo, questo valore è pari a 12. Osserviamo che il gradiente di y è ancora `None` :

In [213]:
```
print(y.grad)
```

```
None
```

Ciò è normale in quanto `y` è la variabile **dipendente**. Notiamo che la differenziazione può essere effettuata solo una volta. Se proviamo a eseguire `backward` nuovamente, otteniamo un errore. Si provi a eseguire nuovamente:

```
y.backward()
```

Attenzione, la funzione `backward` non cancella eventuali vecchi valori dei gradienti, ma li accumula (ovvero li somma). Consideriamo il seguente esempio:

In [214]:
```
x = torch.Tensor([3])
x.requires_grad_(True)
y = 2*x**2

y.backward()
print(x.grad)

y = 2*x**2
y.backward()
print(x.grad)
```

```
tensor([12.])
tensor([24.])
```

Il gradiente di y rispetto a x è sempre 12, ma nel secondo caso, questo valore viene sommato al gradiente pre-esistente. Per evitare di accumulare i gradienti, è necessario azzerarli manualmente come segue:

In [215]:
```
x = torch.Tensor([3])
x.requires_grad_(True)
y = 2*x**2

y.backward()
print(x.grad)

x.grad.data.zero_()

y = 2*x**2
y.backward()
print(x.grad)
```

```
tensor([12.])
tensor([12.])
```

Domanda 12

Si consideri il seguente codice

```
x = Variable(torch.Tensor([6]), requires_grad=True)
y = 0.5 * x**2
y.backward()
```

Qual è il valore di `x.grad` ?

Risposta 12

Domanda 13

Il seguente codice viene eseguito dopo il precedente:

```
y = x**2
y.backward()
```

Qual è il valore di `x.grad` ?

Risposta 13

4.2.1 Esempio a più variabili

Generalmente si parla di gradiente nel caso di funzioni di più variabili indipendenti. In questi casi, il gradiente può essere definito come un vettore che ha come componenti le derivate parziali di una funzione rispetto alle singole variabili. Ad esempio, consideriamo la seguente funzione di tre variabili indipendenti x, y e z:

$$f(x, y, z) = \frac{x^2 - \frac{1}{y}}{z}$$

Il gradiente ∇f di f è una funzione vettoriale. I suoi valori sono dunque vettori di tre dimensioni. In particolare, se assumiamo un sistema di riferimento Cartesiano ortonormale (ogni terna di valori (x, y, z) può essere vista come un punto nello spazio 3D), il gradiente di f può essere definito come:

$$\nabla f = \frac{\partial f}{\partial x}\hat{\mathbf{x}} + \frac{\partial f}{\partial y}\hat{\mathbf{y}} + \frac{\partial f}{\partial z}\hat{\mathbf{z}}$$

dove $\frac{\partial f}{\partial x}$, $\frac{\partial f}{\partial y}$ e $\frac{\partial f}{\partial z}$ sono le derivate parziali di f rispetto a x, y e z e $\hat{\mathbf{x}}$, $\hat{\mathbf{y}}$ e $\hat{\mathbf{z}}$ sono i versori lungo i tre assi.

Pertanto, il gradiente di f calcolato nel punto (x, y, z) sarà dato da:

$$\nabla f(x, y, z) = (\frac{\partial f}{\partial x}(x, y, z), \frac{\partial f}{\partial y}(x, y, z), \frac{\partial f}{\partial z}(x, y, z))$$

Calcoliamo le derivate parziali della funzione f:

$$\frac{\partial f}{\partial x} = \frac{2x}{z}, \frac{\partial f}{\partial y} = \frac{1}{y^2 z}, \frac{\partial f}{\partial y} = \frac{1 - x^2 y}{z^2 y}$$

Qual è il gradiente di f nel punto (3,2,3)?

$$\nabla f(3, 2, 3) = (\frac{\partial f}{\partial x}(3, 2, 3), \frac{\partial f}{\partial y}(3, 2, 3), \frac{\partial f}{\partial z}(3, 2, 3)) = (2, 0.08, -0.95)$$

In pratica, PyTorch ci permette di ottenre lo stesso risultato senza dover specificare esplicitamente le derivate parziali.

Definiamo la funzione f:

In [216]:
```
f = lambda x,y,z : (x**2-1/y)/z
```

Adesso definiamo tre variabili indipendenti x, y e z e assegniamo loro i valori 3,2,3:

In [217]:
```
x = torch.Tensor([3]); x.requires_grad_(True)
y = torch.Tensor([2]); y.requires_grad_(True)
z = torch.Tensor([3]); z.requires_grad_(True)
```

Out[217]:
```
tensor([3.], requires_grad=True)
```

Applichiamo la funzione f alle tre variabili:

In [218]:
```
result = f(x,y,z)
```

`result` è una nuova variabile:

In [219]:
```
print(result)
```

```
tensor([2.8333], grad_fn=<DivBackward0>)
```

Avviamo il processo di differenziazione richiamando `backward` sulla variabile indipendente:

In [220]:
```
result.backward()
```

Controlliamo i valori dei gradienti rispetto a x, y e z:

In [221]:
```
print("%0.2f" % x.grad)
print("%0.2f" % y.grad)
print("%0.2f" % z.grad)
```

```
2.00
0.08
-0.94
```

Domanda 14

Si consideri il seguente codice:

```
x = Variable(torch.Tensor([3]), requires_grad=True)
y = Variable(torch.Tensor([6]), requires_grad=True)

z = 0.5*x**2+3*y
z.backward()
```

Quali sono i valori di `x.grad` e `y.grad` ?

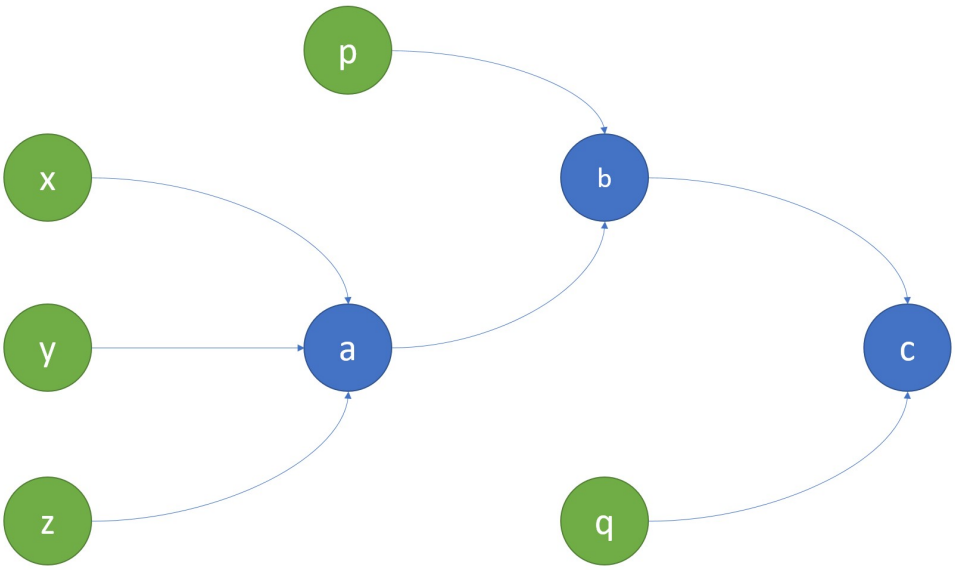
Risposta 14

4.2.2 Grafo computazionale

Tensori e operazioni tra tensori formano un grafo diretto aciclico. Se tutte le operazioni sono differenziabili, autograd permette di calcolare in automatico i gradienti di ogni nodo del grafi rispetto al nodo sul quale si è richiamato il metodo `backward` . Vediamo un esempio:

In [222]:
```
x=torch.Tensor([1])
y=torch.Tensor([5])
z=torch.Tensor([2])
p=torch.Tensor([5])
q=torch.Tensor([6])

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)
p.requires_grad_(True)
q.requires_grad_(True)

a=x+2*y+3*z
b=a*p
c=b/q
```

I tensori e le operazioni definiscono il seguente grafo diretto aciclico:



Nel grafo, i nodi in verde identificano variabili definite dall'utente (variabili indipendenti), mentre i nodi blu identificano variabili che sono frutto di operazioni tra tensori (variabili dipendenti).

Dato un nodo (es. c), autograd permette di calcolare automaticamente le derivate parziali dei nodi creati dall'utente (variabili indipendenti) rispetto al nodo scelto (variabili dipendenti). Ad esempio, richiamando il metodo:

```
In [223]: c.backward()
```

Otteniamo le seguenti derivate parziali:

$$\frac{\partial c}{\partial x}(1)$$

```
In [224]: x.grad
Out[224]: tensor([0.8333])
```

$$\frac{\partial c}{\partial y}(5)$$

```
In [225]: y.grad
Out[225]: tensor([1.6667])
```

$$\frac{\partial c}{\partial z}(2)$$

```
In [226]: z.grad
Out[226]: tensor([2.5000])
```

$$\frac{\partial c}{\partial p}(5)$$

```
In [227]: p.grad
Out[227]: tensor([2.8333])
```

$$\frac{\partial c}{\partial q}(6)$$

```
In [228]: q.grad
Out[228]: tensor([-2.3611])
```

4.2.2 Fermare il calcolo dei gradienti

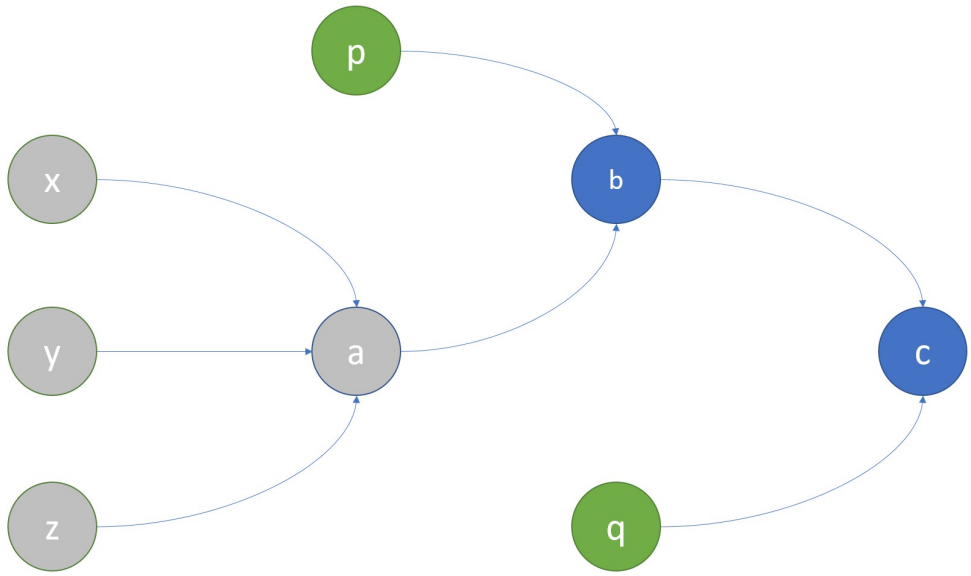
Tensori e funzioni formano un grafo diretto aciclico. Richiamando `backward` su uno dei nodi, verrà attivato il processo di differenziazione automatico che coinvolgerà in cascata tutti i nodi ad esso connesso. Nel caso di grafi molto grandi, il calcolo dei gradienti può essere una operazione computazionalmente costosa. Nel caso in cui non serva calcolare i gradienti rispetto ad un determinato tensore, è possibile fermarne il calcolo in maniera esplicita.

Detach

Il metodo `detach` permette di "staccare" un tensore dal grafo diretto aciclo, di fatto escludendo la computazione dei gradienti dei nodi del sottografo "staccato". Riprendiamo l'esempio precedente, ma questa volta calcoliamo il valore di `b` utilizzando una versione "detached" di `a` :

In [229]:
```
x=torch.Tensor([1])
y=torch.Tensor([5])
z=torch.Tensor([2])
p=torch.Tensor([5])
q=torch.Tensor([6])

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)
p.requires_grad_(True)
q.requires_grad_(True)

a=x+2*y+3*z
b=a.detach()*p ###NOTA a.detach()
c=b/q
```

Il codice scritto sopra è equivalente al seguente grafo.



Dove per i nodi in grigio non viene calcolato alcun gradiente. Verifichiamo:

In [230]:
```
c.backward()
print(x.grad)
print(y.grad)
print(z.grad)
print(p.grad)
print(q.grad)
```

```
None
None
None
tensor([2.8333])
tensor([-2.3611])
```

Come ci aspettavamo, i gradienti relativi a `x` , `y` e `z` non sono stati calcolati, mentre quelli relativi a `p` e `q` sono stati correttamente calcolati.

with torch.no_grad()

E' inoltre possibile disattivare o attivare la costruzione stessa del grafo computazionale inserendo il codice in uno specifico contesto attivato mediante `with torch.no_grad()` . Vediamo un esempio in cui la costruzione del grafo è abilitata:

In [231]:
```
x=torch.Tensor([1])
y=torch.Tensor([5])
z=torch.Tensor([2])
p=torch.Tensor([5])
q=torch.Tensor([6])

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)
p.requires_grad_(True)
q.requires_grad_(True)

#specificando "True", abilitiamo la costruzione del grafo
with torch.set_grad_enabled(True):
    a=x+2*y+3*z
    b=a*p
    c=b/q

    c.backward()
    print(x.grad)
    print(y.grad)
    print(z.grad)
    print(p.grad)
    print(q.grad)
```

```
tensor([0.8333])
tensor([1.6667])
tensor([2.5000])
tensor([2.8333])
tensor([-2.3611])
```

Analogamente, possiamo disabilitare la costruzione del grafo come segue:

In [232]:
```
x=torch.Tensor([1])
y=torch.Tensor([5])
z=torch.Tensor([2])
p=torch.Tensor([5])
q=torch.Tensor([6])

x.requires_grad_(True)
y.requires_grad_(True)
z.requires_grad_(True)
p.requires_grad_(True)
q.requires_grad_(True)

#specificando "False", disabilitiamo la costruzione del grafo
with torch.set_grad_enabled(False):
    a=x+2*y+3*z
    b=a*p
    c=b/q

    try:
        c.backward() #questa operazione scatena una eccezione, il calolo dei gradienti è disabilitato
    except Exception as e:
        print(e)
    print(x.grad)
    print(y.grad)
    print(z.grad)
    print(p.grad)
    print(q.grad)
```

```
element 0 of tensors does not require grad and does not have a grad_fn
None
None
None
None
None
```

In questo secondo esempio, notiamo come, sebbene il calcolo dei gradienti sia abilitato per ciascun tensore, il contesto `torch.set_grad_enabled` permette di disabilitare il calcolo dei gradienti e risparmiare memoria.

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 estragga 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

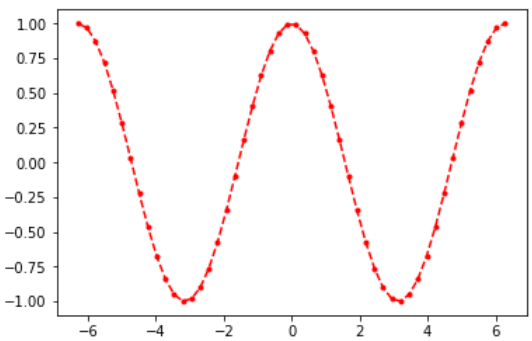
Si generi la seguente matrice scrivendo la minore quantità di codice possibile:

```
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

Definire i seguenti tensori di interi:

- `A: [[1,3,6],[4,5,9]]`
- `B: [[6,7,9],[9,8,2]]`

Si effettuino le seguenti operazioni:

- Calolare la somma tra A e B e mettere il risultato in nuovo tensore C;
- Sottrarre B and A (modificandone il valore);
- Calcolare il massimo per righe di A;
- Calcolare la somma per righe di B;

Esercizio 12

Si consideri la seguente funzione:

$$f(x, y) = \frac{x}{y^x} - y^3$$

Si calcoli il gradiente della funzione nel punto $(0, 8)$

References

- Documentazione di Python 3. https://docs.python.org/3/
- Documentazione di Numpy. http://www.numpy.org/
- Documentazione di PyTorch. https://pytorch.org/docs/stable/index.html
- PyTorch Tensor Tutorial. http://pytorch.org/tutorials/beginner/blitz/tensor_tutorial.html
- PyTorch Autograd Tutorial. http://pytorch.org/tutorials/beginner/blitz/autograd_tutorial.html