Python ML (Machine Learning)

Stefano Di Lena

2024/2025

Contents

1 Introduzione

Python è un ottimo linguaggio di programmazione general-purpose già da sè, ma con l'utilizzo di alcune librerie famose (numpy, scipy, matplotlib) esso diventa un potente ambiente per il calcolo scientifico.

1.1 Basic of Python

Il codice Python è spesso simile allo pseudocodice, dal momento che ti permette di esporre idee in poche linee di codice leggibili.

Di seguito un esempio sull'implementazione del classico algoritmo quicksort:

```
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quicksort(left) + middle + quicksort(right)

print(quicksort([3,6,8,10,1,2,1]))
```

In output riceviamo il vettore ordinato:

```
[1, 1, 2, 3, 6, 8, 10]
```

1.2 Basic data types

1.2.1 Numbers

Integers e floats funzionano come per gli altri linguaggi.

```
    x = 3
    print(x, type(x))
```

Il risultato di questa operazione è:

```
3 <class 'int'>
```

Consideriamo alcune operazioni base tra i numeri:

```
    print(x + 1)   #Addizione
    print(x - 1)   #Sottrazione
    print(x * 2)   #Moltiplicazione
    print(x ** 2) #Esponenziale
```

Output:

```
4
2
6
9
```

Alcune operazioni possono essere scritte anche in modo alternativo:

```
x += 1
print(x)
x *= 2
print(x)
```

Output:

```
4
8
```

In Python non serve definire la classe a prescindere (float o int) ma la prende in auomatico dal numero inserito in input.

```
y = 2.5
print(type(y))
print(y, y + 1, y * 2, y ** 2)
```

Output:

```
<class 'float'>
2.5 3.5 5.9 6.25
```

1.2.2 Booleans

```
t, f = True, False
print(type(t))
```

Output:

```
<class 'bool'>
```

In Python gli operatori vengono implementati utilizzando parole inglesi al posto dei classici simboli (&&, ||, etc.):

```
print(t and f) #AND logico
print(t or f)  #OR logico
print(not t)   #NOT logico
print(t != f)  #XOR logico
```

Output:

```
False
True
False
True
```

1.2.3 Strings

```
hello = 'hello'     #Strings litterals can use sngle quotes
world = "world"     #or double quotes; it does not matter
print(hello, len(hello))
```

In uscita avremo la scritta hello e la lunghezza di tale parola:

```
hello 5
```

```
hw = hello + ' ' + world  #String concatenation
print(hw)
```

Output:

```
hello world
```

Possiamo inoltre formattare le stringhe come segue:

```
hw12 = '{} {} {}'.format(hello, world, 12)
print(hw12)
```

Output:

```
hello world 12
```

Le stringhe hanno anche altri metodi utili; ad esempio:

```
s = "hello"
print(s.capitalize()) #Capitalize a string
print(s.upper()) #Convert a string to uppercase
print(s.rjust(7)) #Right-justify a string (padding with spaces)
print(s.center(7))     #Center a string, padding with spaces
print(s.replace('l', '(ell)')) #Replace all intances of one
    substring with another
print('  world '.strip()) #Strip leading and trailing
    whitespace
```

Output:

```
Hello
HELLO
    hello
 hello
he(ell)(ell)o
world
```

1.3 Containers

Python include diversi tipi di container built-in (lists, dictionaries, sets e tuples).

1.3.1 Lists

Una lista in Python è equivalente ad un array, ma essa può essere ridimenzionata e può contenere elementi di tipi differenti.

```
xs = [3, 1, 2] #Crea una lista
print(xs, xs[2])
print(xs[-1]) #Gli indici negativi contano dalla fine della
    lista (a schermo avremo l'elemento 2)
```

In output:

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

```
xs[2] = 'foo' #Una lista potrebbe contenere anche elementi di
    tipi differenti
print(xs)
```

Output:

```
[3, 1, 'foo']
```

Possiamo facilmete aggiungere un elemento ad una lista:

```
xs.append('bar') #Aggiunge un nuovo elemento alla fine della
    lista
print(xs)
```

Output:

```
[3, 1, 'foo', 'bar']
```

Altrettanto semplice è rimuovere un elemento da una lista:

```
x = xs.pop() #Rimuove e  restituisce l'ultimo elemento della
    lista
print(x, xs)
```

Output:

```
bar [3, 1, 'foo']
```

1.3.2 Slicing

Con questo termine si intende l'accesso a sublists tramite sintassi concisa.

```
nums = list(range(5)) #la funzione range e' una funzione built-
    in che crea una lista di integers
print(nums)
print(nums[2:4]) #Stampa soltanto i numeri all'interno dello
    slice dall'indice 2 al 4
print(nums[2:]) #Stampa uno slice dall'indice 2 alla fine
print(nums[:2]) #Stampa i numeri dall'inizio all'indice 2
print(nums[:]) #Stampa uno slide dell'intera l'ista
print(nums[:-1])
nums[2:4] = [8, 9] #Assegna una nuova sottolista ad uno slice
print(nums) #Stampa la nuova lista
```

Output:

```
[0, 1, 2, 3, 4]
[2, 3]
[2, 3, 4]
[0, 1]
[0, 1, 2, 3, 4]
[0, 1, 2, 3]
[0, 1, 8, 9, 4]
```

1.3.3 Loops

```
animals = ['cat', 'dog', 'monkey']
for animal in animals:
    print(animal)
```

Output:

```
cat
dog
monkey
```

Possiamo accedere anche agli indici di ogni elemento utilizzando le funzioni di enumerazione built-in.

```
animal = ['cat', 'dog', 'monkey']
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(inx + 1, animal))
```

Output:

```
#1: cat
#2: dog
#3: monkey
```

1.3.4 List comprehensions

Soltamente, quando programmiamo, vogliamo trasformare un tipo di dato in un altro.

```
num = [0, 1, 2, 3, 4]
squares = []
for x in nums
    squares.append(x ** 2)
print(squares)
```

Output:

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

Questo è semplificabile usando una list comprehension:

```
nums = [0, 1, 2, 3, 4]
squares = [x ** 2 for x in nums]
print(squares)
```

Output:

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

Le list comprehension possono contenere anche delle condizioni, ad esempio scegliamo solamente le radici divisibili per 2:

```
nums = [0, 1, 2, 3, 4]
even_squares = [x ** 2 for x in nums if x % 2 == 0]
print(even_squares)
```

Output:

```
[0, 4, 16]
```

1.3.5 Dictionaries

Un dizionario conserva delle coppia (key, value).

```
d = {'cat': 'cute', 'dog': 'furry'} #Crea un nuovo dizionario
    con alcuni dati
print(d['cat']) #Stampa una voce dal dizionario
print('cat' in d) #Controlla se il dizionario ha una chiave
    data (in uscita otteremo un valore booleano)
```

Output:

```
cute
True
```

Possiamo facilmente aggiungere parole al dizionario.

```
d['fish'] = 'wet'
print(d['fish'])
```

Output:

```
wet
```

Se provimo a stampare un elemento non appartenente al dizionario:

```
print(d['monkey']) #Otteniamo un KeyError: 'monkey' not a kay
    of d
```

Otterremo il seguente errore:

```
KeyError                         Traceback (most recent call last)
<ipython-input-25-78fc9745d9cf> in <module>
----> 1 print(d['monkey'])  # KeyError: 'monkey' not a key of d

KeyError: 'monkey'
```

Possiamo risolvere il problema in questo modo:

```
print(d.get('monkey', 'N/A'))
print(d.get('fish', 'N/A'))
```

Così facendo in output stamperà N/A ogni qualvolta un elemento non è stato trovato all'interno del dizionario.

```
N/A
wet
```

Un elemento può essere facilmente cancellato da un dizionario.

```
del d['fish']
print(d.get('fish', 'N/A'))
```

Output:

```
N/A
```

É facile in un dizionario iterare sulle chiavi.

```
d = {'person': 2, 'cat': 4, 'spider': 8}
for animal, legs in d.items():
    print('A {} has {} legs'.format(animal, legs))
```

Output:

```
A person has 2 legs
A cat has 4 legs
A spider has 8 legs
```

Eistono anche dictionary comprehension che ti permettono facilmente di costruire un dizionario.

```
nums = [0, 1, 2, 3, 4]
even_num_to_square = {x: x** 2 for x in nums if x % 2 == 0}
print(even_num_to_square
```

Output:

```
{0: 0, 2: 4, 4: 16}
```

1.3.6 Sets

A set è una collezione disordinata di elementi distinti.

```
animals = {'cat', 'dog'}
print('cat' in animals) #Controlla se l'elemento e' presente
    nel set (restituisce un valore booleano)
print('fish' in animals)
```

Output:

```
True
False
```

Possiamo facilmente aggiungere elementi ad un set:

```
animals.add('fish')
print('fish' in animals)
print(len(animals)) #stampa la lunghezza del set (numeo di
    elementi presenti in esso)
```

Output:

```
True
3
```

```
animal.add('cat') #Aggiungere un elemento gia' presente all'
    interno di un set non fa' nulla
print(len(animals))
animals.remove('cat') #Rimuove un elemento dal set
print(len(animals))
```

Output:

```
3
2
```

Le iterazioni su un set hanno la stessa sintassi di quelle su una lista, ma dato che i set non hanno un ordine non puoi fare assunzioni sull'ordine in cui vengono presi gli elementi.

```
animals = {'cat', 'dog', 'fish'}
for idx, animal in enumerate(animals):
    print('#{}: {}'.format(idx + 1, animal))
```

Output:

```
#1: fish
#2: cat
#3: dog
```

Possiamo facilmente costruire dei sets usando set comprehensions.

```
from math import sqrt
print({int(sqrt(x)) for x in range(30)})
```

Output:

```
0, 1, 2, 3, 4, 5
```

1.3.7 Tuples

Una tupla è un immutabile lista ordinata di valori. Essa è molto simile alla lista; una delle principali differenze è quella che le tuple possono essere utilizzata come keys nei dizionari e come elementi di un set mentre le liste no.

```
d = {(x, x+1): x for x in range(10)} #Crea un dizionare in cui
    le chiavi sono tuple
t = (5, 6) #Crea una tupla
print(type(t))
print(d[t])
print(d[(1,2)])
```

Output:

```
{(0, 1): 0, (1, 2): 1, (2, 3): 2, (3, 4): 3, (4, 5): 4,
(5,6): 5, (6, 7): 6, (7, 8): 7, (8, 9): 8, (9, 10): 9}
<class 'tuple'>
5
1
```

```
t[0] = 1
```

Output:

```
TypeError            Traceback (most recent call last)
<ipython-input-38-c8aeb8cd20ae> in <module>
----> 1 t[0] = 1

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

1.4 Functions

Le funzioni in Python sono definite utilizzando il keyword "def".

```
def sign(x):
    if x > 0:
        return 'positive'
    elif x < 0:
        return 'negative'
    else:
        return 'zero'

for x in [-1, 0, 1]:
    print(sign(x))
```

Output:

```
negative
zero
positive
```

Spesso definiamo le funzioni che prenano in input argomenti keyword come parametri opzionali.

```
def hello(name, loud = False):
    if loud:
        print('HELLO, {}'.format(name.upper()))
    else:
        print('Hello, {}!'.format(name)

hello('Bob')
hello('Fred', loud = True)
```

Output:

```
Hello, Bob!
HELLO, FRED
```

1.5 Classes

```
class Greeter:

    #Constructor
    def __init__(self, name):
        self.name = name #Create an instance variable

    ##Instance method
    def greet(self, loud=False):
        if loud:
            print('HELLO, {}'.format(self.name.upper()))
        else:
            print('Hello, {}!'.format(self.name))

g = Greeter('Fred') #Construct an instance of the Greeter class
g.greet() #Call an instance method
g.greet(loud=True) #Call an instance method
```

Output:

```
Hello, Fread!
HELLO, FRED
```

2 Numpy

Numpy è una lbreria utilizzatissima nel calcolo scientifico in Python. Questa provvede molti tools per lavorare con array multidimensionali. Per utilizzare Numpy dobbiamo prima di tutto importare il package:

```
import numpy as np
```

2.1 Arrays

Un array numpy è una griglia di valori, tutti dello stesso tipo, che sono indicizzati da una tupla di integer non-negaivi.

Possiamo inizializzare un array numy da una lista Python annidata ed accedere agli elementi utilizzado le parentesi quadre.

```
a = np.array([1, 2, 3]) #Crea un array di rank 1
print(type(a), a.shape, a[0], a[1], a[2])
a[0] = 5 #Cambiare un elemento di un array
print(a)
```

Output:

```
<class 'numpy.ndarray'> (3,) 1 2 3
[5 2 3]
```

```
b = np.array([[1,2,3],[4,5,6]]) #Crea un array di rank 2
print(b)
```

Output:

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

```
print(b.shape)
print(b[0, 0], b[0, 1], b[1, 0])
```

Output:

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

Numpy fornisce inoltre delle funzioni per crearre gli arrays:

```
a = np.zeros((2,2)) #Crea un array di zero
print(a)
```

Output:

```
[[0. 0.]
 [0. 0.]]
```

```
b = np.ones((1,2))
print(b)
```

Output:

```
[[1. 1.]]
```

```
c = np.full((2,2), 7) #Crea un array costante
print(c)
```

Output:

```
[[7 7]
 [7 7]]
```

```
d = np.eye(2) #Crea una matrice Identita' 2x2
print(d)
```

Output:

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

```
e = np.random.random((2,2)) #Crea un array di valori casuali
print(e)
```

Un esempio di output potrebbe essere:

```
[[0.40850108 0.11156029]
 [0.12665546 0.28954288]]
```

2.1.1 Array indexing

Numpy offre diversi modi per rifersi agli elementi dell'array.

```
import numpy as np

#Create the following rank 2 array with shape (3, 4)
# [[ 1  2  3  4]
#  [ 5  6  7  8]
#  [ 9 10 11 12]]
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])

#Use slicing to pull out the subarray consisting of the first
#2 rows
#and columns 1 and 2; b is the following array of shape (2, 2):
# [[2 3]
#  [6 7]]
b = a[:2, 1:3]
print(b)
```

Output:

```
[[2 3]
 [6 7]]
```

```
print(a[0, 1])
b[0, 0] = 77 #b[0, 0] e' lo stesso pezzo di dati di a[0, 1]
print(a[0, 1])
```

Output:

```
2
77
```

```
#Create the following rank 2 array with shape (3, 4)
a = np.array([[1,2,3,4], [5,6,7,8], [9,10,11,12]])
print(a)
```

Output:

```
[[ 1 2 3 4]
 [ 5 6 7 8]
 [ 9 10 11 12]]
```

```
row_r1 = a[1, :]    #Rank 1 view of the second row of a
row_r2 = a[1:2, :]  #Rank 2 view of the second row of a
row_r3 = a[[1], :]  #Rank 2 view of the second row of a
print(row_r1, row_r1.shape)
print(row_r2, row_r2.shape)
print(row_r3, row_r3.shape)
```

Output:

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

```
#We can make the same distinction when accessing columns of an
    array:
col_r1 = a[:, 1]
col_r2 = a[:, 1:2]
print(col_r1, col_r1.shape)
print()
print(col_r2, col_r2.shape)
```

Output:

```
[ 2  6 10] (3,)

[[ 2]
 [ 6]
 [10]] (3, 1)
```

L'array indexing ti permette di costruire arrays arbitrari usando dati di altri array.

```
a = np.array([[1,2], [3, 4], [5, 6]])

#An example of integer array indexing.
#The returned array will have shape (3,) and
print(a[[0, 1, 2], [0, 1, 0]])

#The above example of integer array indexing is equivalent to
    this:
print(np.array([a[0, 0], a[1, 1], a[2, 0]]))
```

Output:

```
[1 4 5]
[1 4 5]
```

```
#When using integer array indexing, you can reuse the same
#element from the source array:
print(a[[0, 0], [1, 1]])

#Equivalent to the previous integer array indexing example
print(np.array([a[0, 1], a[0, 1]]))
```

Output:

```
[2 2]
[2 2]
```

Possiamo inoltre mutare o selezionare un elemento da ogni righa della matrice.

```
#Create a new array from which we will select elements
a = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
print(a)
```

Output:

```
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

```
#Create an array of indices
b = np.array([0, 2, 0, 1])

#Select one element from each row of a using the indices in b
print(a[np.arange(4), b])
```

Output:

```
[ 1  6  7 11]
```

```
#Mutate one element from each row of a using the indices in b
a[np.arange(4), b] += 10
print(a)
```

Output:

```
[[11  2  3]
 [ 4  5 16]
 [17  8  9]
 [10 21 12]]
```

```
import numpy as np

a = np.array([[1,2], [3, 4], [5, 6]])

bool_idx = (a > 2)  #Find the elements of a that are bigger
    than 2; this returns a numpy array of Booleans of the same
    shape as a, where each slot of bool_idx tells whether that
    element of a is > 2.

print(bool_idx)
```

Output:

```
[[False False]
 [ True  True]
 [ True  True]]
```

```
#We use boolean array indexing to construct a rank 1 array
    consisting of the elements of a corresponding to the True
    values of bool_idx
print(a[bool_idx])

#We can do all of the above in a single concise statement:
print(a[a > 2])
```

Output:

```
[3 4 5 6]
[3 4 5 6]
```

2.1.2 Datatypes

Ogni array numpy è una griglia di elementi dello stesso tipo. Numpy provvede un grande set di datatypes numerici che possono essere usati per costruire gli arrays. Numpy prova ad indovinare il datatype quando tu crei un nuovo array, ma le funzioni per costruire gli array spesso includono un argomento opzionale per specificare esplicitamente il datatype.

```
x = np.array([1, 2])  #Let numpy choose the datatype
y = np.array([1.0, 2.0])  # Let numpy choose the datatype
z = np.array([1, 2], dtype=np.int64)  # Force a particular
    datatype

print(x.dtype, y.dtype, z.dtype)
```

Output:

```
int64 float64 int64
```

2.1.3 Array math

Nei moduli numpy sono disponibili delle funzioni matematiche che operano elemento per elemento sugli array.

```
x = np.array([[1,2],[3,4]], dtype=np.float64)
y = np.array([[5,6],[7,8]], dtype=np.float64)

#Elementwise sum; both produce the array
print(x + y)
print(np.add(x, y))
```

Output:

```
[[ 6.  8.]
 [10. 12.]]
[[ 6.  8.]
 [10. 12.]]
```

```
#Elementwise difference; both produce the array
print(x - y)
print(np.subtract(x, y))
```

Output:

```
[[-4. -4.]
 [-4. -4.]]
[[-4. -4.]
 [-4. -4.]]
```

```
#Elementwise product; both produce the array
print(x * y)
print(np.multiply(x, y))
```

Output:

```
[[ 5. 12.]
 [21. 32.]]
[[ 5. 12.]
 [21. 32.]]
```

```
# Elementwise division;
print(x / y)
print(np.divide(x, y))
```

Output:

```
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
[[0.2        0.33333333]
 [0.42857143 0.5       ]]
```

```
#Elementwise square root;
print(np.sqrt(x))
```

Output:

```
[[1.         1.41421356]
 [1.73205081 2.        ]]
```

A differenza di MATLAB il * rappresenta la moltiplicazione elemento per elemento e non quella tra matrici (riga per colonna). Per effettuare il prodotto tra matrici si usa la funzione dot.

```
x = np.array([[1,2],[3,4]])
y = np.array([[5,6],[7,8]])

v = np.array([9,10])
w = np.array([11, 12])

#Inner product of vectors
print(v.dot(w))
print(np.dot(v, w))
```

Output:

```
219
219
```

Si può usare anche l'operatore @ per ottenere lo stesso risoltato.

```
#Matrix / vector product
print(x.dot(v))
print(np.dot(x, v))
print(x @ v)
```

Output:

```
[29 67]
[29 67]
[29 67]
```

```
# Matrix / matrix product
print(x.dot(y))
print(np.dot(x, y))
print(x @ y)
```

Output:

```
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
[[19 22]
 [43 50]]
```

Numpy provvede, inoltre, molte funzioni utili per la computazione degli array.

```
x = np.array([[1,2],[3,4]])

print(np.sum(x))  #Compute sum of all elements
print(np.sum(x, axis=0))  #Compute sum of each column
print(np.sum(x, axis=1))  #Compute sum of each row
```

Output:

```
10
[4 6]
[3 7]
```

Molte volte è necessario manipolare i dati degli array. Ad esempio, per effettuare la trasposta di una matrice usiamo:

```
print(x)
print("transpose\n", x.T)
```

Output:

```
[[1 2]
 [3 4]]
transpose
[[1 3]
 [2 4]]
```

```
v = np.array([[1,2,3]])
print(v )
print("transpose\n", v.T)
```

Output:

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

2.1.4 Broadcasting

Un meccanismo potente che permette a numpy di lavorare con gli array di forme differenti (quando performa operazioni aritmetiche) è il Broadcasting. Molte volte abbiamo un array più piccolo ed uno più grande e vogliamo utilizzare l'array minore più volte per operare sull'array maggiore.

```
#We will add the vector v to each row of the matrix x, storing the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = np.empty_like(x)   #Create an empty matrix with the same shape as x

#Add the vector v to each row of the matrix x with an explicit loop
for i in range(4):
    y[i, :] = x[i, :] + v

print(y)
```

Output:

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Se la matrice **x** fosse troppo grande, computare in un loop potrebbe essere una soluzione lenta. Aggiungendo il vettore **v** in ogni riga di **x** è eguivalente a formare una matrice **vv** creando uno stack di copie di **v** verticalmente e successivamente effettuare la somma per elementi tra **x** e **vv**.

```
vv = np.tile(v, (4, 1))  #Stack 4 copies of v on top of each other
print(vv)
```

Output:

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

```
y = x + vv  #Add x and vv elementwise
print(y)
```

Output:

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Numpy ti permette di effettuare computazioni senza creare realmente più copie di **v**.

```
import numpy as np

#We will add the vector v to each row of the matrix x, storing
    the result in the matrix y
x = np.array([[1,2,3], [4,5,6], [7,8,9], [10, 11, 12]])
v = np.array([1, 0, 1])
y = x + v  #Add v to each row of x using broadcasting
print(y)
```

Output:

```
[[ 2  2  4]
 [ 5  5  7]
 [ 8  8 10]
 [11 11 13]]
```

Il broadcasting tipicamente rende il codice più compatto e veloce, quindi è utile sforzarsi ad implementarlo dove possibile.

```
#Compute outer product of vectors
v = np.array([1,2,3])  #v has shape (3,)
w = np.array([4,5])    #w has shape (2,)
#To compute an outer product, we first reshape v to be a column
     vector of shape (3, 1); we can then broadcast it against w
     to yield an output of shape (3, 2), which is the outer
    product of v and w:

print(np.reshape(v, (3, 1)) * w)
```

Output:

```
[[ 4  5]
 [ 8 10]
 [12 15]]
```

```
#Add a vector to each row of a matrix
x = np.array([[1,2,3], [4,5,6]])
#x has shape (2, 3) and v has shape (3,) so they broadcast to
    (2, 3), giving the following matrix:

print(x + v)
```

Output:

```
[[2 4 6]
 [5 7 9]]
```

```
#Add a vector to each column of a matrix x has shape (2, 3) and
     w has shape (2,). If we transpose x then it has shape (3,
    2) and can be broadcast against w to yield a result of
    shape (3, 2); transposing this result yields the final
    result of shape (2, 3) which is the matrix x with the
    vector w added to each column. Gives the following matrix:

print((x.T + w).T)
```

Output:

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
#Another solution is to reshape w to be a row vector of shape
    (2, 1); we can then broadcast it directly against x to
    produce the same output.
print(x + np.reshape(w, (2, 1)))
```

Output:

```
[[ 5  6  7]
 [ 9 10 11]]
```

```
# Multiply a matrix by a constant: x has shape (2, 3). Numpy
    treats scalars as arrays of shape (); these can be
    broadcast together to shape (2, 3), producing the following
     array:
print(x * 2)
```

Output:

```
[[ 2  4  6]
 [ 8 10 12]]
```

3 Matplotlib

Matplotlib è una libreria per mostrare grafici, molto simile a quella di MATLAB.

```
import matplotlib.pyplot as plt
```

```
#By running this special iPython command, we will be displaying
#plots inline
%matplotlib inline
```

3.1 Plotting

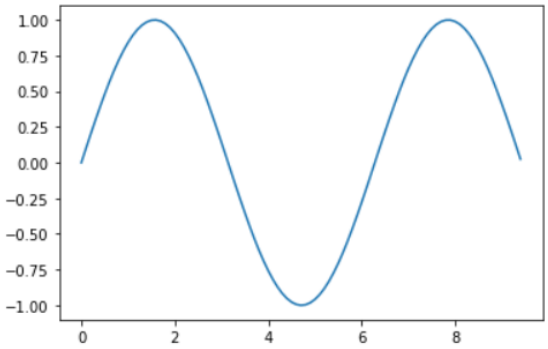
La funzione più importante di matplotlib premette di mostrare dati 2D.

```
#Compute the x and y coordinates for points on a sine curve
x = np.arange(0, 3 * np.pi, 0.1)
y = np.sin(x)

#Plot the points using matplotlib
plt.plot(x, y)
```

Output:

```
[<matplotlib.lines.Line2D at 0x7fb4d1580490>]
```



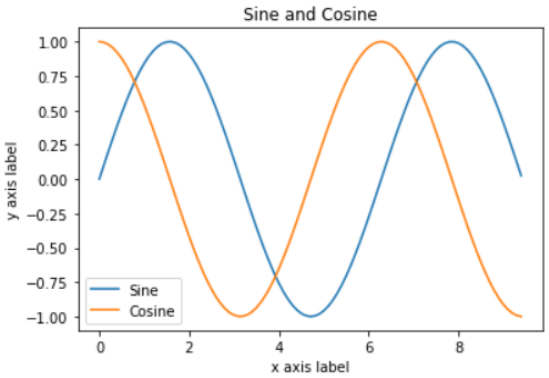
Possiamo anche plottare più righe insieme, aggiungere una leggenda, un titolo e nominare gli assi.

```
y_sin = np.sin(x)
y_cos = np.cos(x)

#Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])
```

Output:

```
<matplotlib.legend.Legend at 0x7fb4d104efd0>
```



3.2 Subplots

Utilizzando la funzione subplot si possono plottare diverse cose nella stessa fiura.

```
#Compute the x and y coordinates for points on sine and cosine
#curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

#Set up a subplot grid that has height 2 and width 1,
#and set the first such subplot as active.
plt.subplot(2, 1, 1)

#Make the first plot
plt.plot(x, y_sin)
plt.title('Sine')

#Set the second subplot as active, and make the second plot.
plt.subplot(2, 1, 2)
plt.plot(x, y_cos)
plt.title('Cosine')

#Show the figure.
plt.show()
```

Output: