Modulo numpy

El modulo numpy es parte del paquete de Python scipy, que permite agregar al lenguaje de programación capacidades de calculo científico, incluyendo un objeto nuevo: el arreglo.

In [2]:

```
import numpy as np
```

Arreglos unidimensionales (vectores)

Se define un arreglo simple con una lista

In [3]:

```
# El arreglo a a partir de una lista
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,])
# Se verifica que no es una lista sino un arreglo
print(type(a))
# Se imprime el contenido del arreglo
print("Array a = ",a)
```

```
<class 'numpy.ndarray'>
Array a =  [0 1 2 3 4 5 6 7 8 9]
```

Se puede acceder a los elementos de un arreglo como si fuera una lista *[inicio:fin:paso] default[0:end-1:1]*

In [4]:

```
print("Primer elemento:", a[0])
print("Ultimo elemento:", a[-1])
print("Tres primeros elementos:", a[:3])
print("Dos ultimos elementos:", a[-2:])
print("Elementos de indice par:", a[1:-1:2])
print("Del elemento 2do al 4to:", a[1:4])
```

```
Primer elemento: 0
Ultimo elemento: 9
Tres primeros elementos: [0 1 2]
Dos ultimos elementos: [8 9]
Elementos de indice par: [1 3 5 7]
Del elemento 2do al 4to: [1 2 3]
```

Se pueden examinar los elementos de un arreglo

In [6]:

```
a = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
print(a)
print("Tamaño del arreglo:", a.size)
print("Dimensiones del arreglo:", a.shape)
print("Tipo de datos:", a.dtype)
print("Cantidad de memoria por dato (bytes):", a.itemsize)
print("Memoria total (bytes):", a.size * a.itemsize)
```

```
[0 1 2 3 4 5 6 7 8 9]
Tamaño del arreglo: 10
Dimensiones del arreglo: (10,)
Tipo de datos: int32
Cantidad de memoria por dato (bytes): 4
Memoria total (bytes): 40
```

Otra forma de crear una arreglo es utilizado el método rango de numpy (np.arange)

In [7]:

```
a = np.arange(10)
print("np.arrange(10) =", a)
a = np.arange(1,10)
print("np.arrange(1,10) =", a)
a = np.arange(1,10,2)
print("np.arrange(1,10,2) =", a)
a = np.arange(10, 0, -1)
print("np.arrange(10,0,-1) =", a)
a = np.arange(1, 10, 0.5)
print("np.arrange(1,10,0.5) =", a)
```

```
np.arrange(10) = [0 1 2 3 4 5 6 7 8 9]
np.arrange(1,10) = [1 2 3 4 5 6 7 8 9]
np.arrange(1,10,2) = [1 3 5 7 9]
np.arrange(10,0,-1) = [10  9  8  7  6  5  4  3  2  1]
np.arrange(1,10,0.5) = [ 1.   1.5  2.   2.5  3.   3.5  4.   4.5  5.   5.5  6.   6.5  7.   7.5  8.
  8.5  9.   9.5]
```

Por defecto, los datos de un ndarray dependeran de los argumentos del método arange. En el ejemplo anterior los datos 1. 1.5 2. hacen referencia a 1.0 1.5 2.0, es decir, son float. Para especificar que se requiere que un valor sea de un tipo especifico con el atributo dtype

In [9]:

```
print(np.arange(1,10, 2, dtype=np.float64))
print(np.arange(1,10, 2, dtype=np.int64))
```

```
[ 1.  3.  5.  7.  9.]
[1 3 5 7 9]
```

A veces es util generar N datos entre dos valores Esto se puede realizar con la método np.linspace(ini, fin, num_elementos)

In [11]:

```
a = np.linspace(0, 10, 5)
print("np.linspace(0, 10, 5) =", a)
a = np.linspace(10, 0, 5)
print("np.linspace(10, 0, 5) =", a)
```

```
np.linspace(0, 10, 5) = [  0.   2.5  5.   7.5 10. ]
np.linspace(10, 0, 5) = [ 10.   7.5  5.   2.5  0. ]
```

Si no se especifica el numero de elementos que se requieren en np.linspace, se generan 50 elementos:

In [13]:
```
a = np.linspace(0, 10)
print("np.linspace(0, 10) =", a)
print("Numero de elementos:", len(a))
```

```
np.linspace(0, 10) = [  0.          0.20408163   0.40816327   0.6122449    0.81632653
   1.02040816   1.2244898    1.42857143   1.63265306   1.83673469
   2.04081633   2.24489796   2.44897959   2.65306122   2.85714286
   3.06122449   3.26530612   3.46938776   3.67346939   3.87755102
   4.08163265   4.28571429   4.48979592   4.69387755   4.89795918
   5.10204082   5.30612245   5.51020408   5.71428571   5.91836735
   6.12244898   6.32653061   6.53061224   6.73469388   6.93877551
   7.14285714   7.34693878   7.55102041   7.75510204   7.95918367
   8.16326531   8.36734694   8.57142857   8.7755102    8.97959184
   9.18367347   9.3877551    9.59183673   9.79591837  10.        ]
Numero de elementos: 50
```

Un arreglo tiene ciertos atributos:

In [14]:
```
print("a = ", a)
print("Numero de elementos:", len(a))
print("Dimensiones:",np.ndim(a))
print("Forma del arreglo:", np.shape(a))
```

```
a =  [  0.          0.20408163   0.40816327   0.6122449    0.81632653
   1.02040816   1.2244898    1.42857143   1.63265306   1.83673469
   2.04081633   2.24489796   2.44897959   2.65306122   2.85714286
   3.06122449   3.26530612   3.46938776   3.67346939   3.87755102
   4.08163265   4.28571429   4.48979592   4.69387755   4.89795918
   5.10204082   5.30612245   5.51020408   5.71428571   5.91836735
   6.12244898   6.32653061   6.53061224   6.73469388   6.93877551
   7.14285714   7.34693878   7.55102041   7.75510204   7.95918367
   8.16326531   8.36734694   8.57142857   8.7755102    8.97959184
   9.18367347   9.3877551    9.59183673   9.79591837  10.        ]
Numero de elementos: 50
Dimensiones: 1
Forma del arreglo: (50,)
```

Se observa que el método np.linspace() incluye en el arreglo generado el numero final, lo que lo distinge de np.arange(). Se puede ajustar esto de forma que no incluya el valor final:

In [15]:
```
a = np.linspace(0, 10, endpoint=False)
print(a)
```

```
[ 0.   0.2  0.4  0.6  0.8  1.   1.2  1.4  1.6  1.8  2.   2.2  2.4  2.6  2.8
  3.   3.2  3.4  3.6  3.8  4.   4.2  4.4  4.6  4.8  5.   5.2  5.4  5.6  5.8
  6.   6.2  6.4  6.6  6.8  7.   7.2  7.4  7.6  7.8  8.   8.2  8.4  8.6  8.8
  9.   9.2  9.4  9.6  9.8]
```

In [4]:
```
#insercion y eliminacion de datos en un arreglo

A = np.array([1, 2, 3, 4, 5, 6])
# Se insertan 0s en un arreglo
A = np.insert(A, 3, [0, 0, 0])   #np.insert(array, pos_index, elements) -> A
print(A)
# Se eliminan los 0s insertados previamente
A = np.delete(A, [3, 4, 5])      #np.delete(array, pos_index, elements) -> A
print(A)
```

```
[1 2 3 0 0 0 4 5 6]
[1 2 3 4 5 6]
```

Por que son importantes los arreglos? Por que se puede hacer con ellos operaciones de elmento-a-elemento

In [16]:

```
lista = range(11)
arreglo = np.arange(11)

'''
for numero in lista
    print(numero**2)
'''

print(list(map(lambda x: x**2, lista)))
print(arreglo**2)

# LAS FUNCIONES MAGICAS SON FUNCIONES QUE SOLO FUNCIONAN EN JUPYTER!!!!
# Utilizemos una "funcion magica" de Python para calcular las diferencias de tiempo de ejecución
# en un calculo mas demandante
lista = range(10000)
%timeit [i**2 for i in lista]

arreglo = np.array(1000)
%timeit arreglo**2
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
[  0   1   4   9  16  25  36  49  64  81 100]
100 loops, best of 3: 5.55 ms per loop
The slowest run took 22.62 times longer than the fastest. This could mean that an intermediate result is being cached.
1000000 loops, best of 3: 777 ns per loop
```

Arreglos bidimensionales (matrices)

Se pueden crear arreglos biimensionales

In [5]:

```
matriz = np.array([[1, 2, 3], [4, 5, 6]])
print(matriz)
print("Numero de elementos (mal):", len(matriz))   # Esto esta mal!!!
print("Numero de elementos:", matriz.size)
print("Dimensiones:",np.ndim(matriz))
print("Forma del arreglo:", matriz.shape)
print("Tipo de datos de los elementos: ", matriz.dtype)  #todos los elementos son iguales
print("Bytes de memoria de los elementos: ", matriz.itemsize)
```

```
[[1 2 3]
 [4 5 6]]
Numero de elementos (mal): 2
Numero de elementos: 6
Dimensiones: 2
Forma del arreglo: (2, 3)
Tipo de datos de los elementos:  int32
Bytes de memoria de los elementos:  4
```

In [6]:

```
A = np.array([[[1, 1, 1], [1, 1, 1]],
              [[2, 2, 2], [2, 2, 2]],
              [[3, 3, 3], [3, 3, 3]]])
print(A)
print("Numero de elementos: ", A.size)
print("Dimensiones de arreglo: ", A.ndim)
print("Rango del arreglo: ", A.shape)
print("Tipo de datos de los elementos: ", A.dtype)  #todos los elementos son iguales
print("Bytes de memoria de los elementos: ", A.itemsize)
```

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

 [[2 2 2]
  [2 2 2]]

 [[3 3 3]
  [3 3 3]]]
Numero de elementos:  18
Dimensiones de arreglo:  3
Rango del arreglo:  (3, 2, 3)
Tipo de datos de los elementos:  int32
Bytes de memoria de los elementos:  4
```

In [7]:

```
# En una matriz, hay que agregar una dimension adicional para la especificacion
# de indices y slicing
A = np.array([[1, 2, 3],
              [4, 5, 6],
              [7, 8, 9]])

print(A[0])         #primera fia
print(A[-1])        #ultima fila
print()
print(A[1][0])      #segunda fila, primera columna
print(A[1][1])      #segunda fila, segunda columna
print()
print(A[0, 0])      #primera fila, primera columna
print(A[-1, 1])     #ultima fila, segunda columna
print()
print(A[1, 1:])     #segunda fila, de la segunda columna hasta la ultima
print(A[:, 1])      #todas las filas, la segunda columna
print(A[:, 1:2])    #conserva la misma forma
print()
print(A[::2,::2])   #elementos en las esquinas
```

```
[1 2 3]
[7 8 9]

4
5

1
8

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

[[1 3]
 [7 9]]
```

In [8]:

```
# Los indices se pueden utilizar para insertar o eliminar filas y columnas,
# pero hay que considerar el argumento axis de un arreglo

print()
A = np.insert(A, 1, [0, 0, 0], 0)      #axis = 0 -> fila
print(A)
print()
A = np.insert(A, 3, [0, 0, 0, 0], 1)   #axis = 1 -> columna
print(A)
print()
A = np.delete(A, 1, axis=0)            #se elimina la fila (axis = 0) idx 1
print(A)
```

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

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

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

In [9]:

```
# Tambien se pueden generar matrices partiendo de vectores (arreglos unidimensionales) y modificando
# la forma del vector
A = np.arange(1, 101).reshape(10, 10)
print(A)
print()
# Y se pueden concatenar arreglos en general especificando el axis
A = np.arange(1,10).reshape(3,3)
A = np.concatenate((A, A), axis=0)
print(A)
print()
A = np.concatenate((A, A), axis=1)
print(A)

# Asi tambien, ciertas operaciones que afectan las filas o columnas
# requieren especificar el axis (sum, cumsum, prod, cumprod)
print()
print(np.sum(A, axis=0))   # Suma por columnas
print(np.sum(A, axis=1))   # Suma por filas
```

```
[[  1   2   3   4   5   6   7   8   9  10]
 [ 11  12  13  14  15  16  17  18  19  20]
 [ 21  22  23  24  25  26  27  28  29  30]
 [ 31  32  33  34  35  36  37  38  39  40]
 [ 41  42  43  44  45  46  47  48  49  50]
 [ 51  52  53  54  55  56  57  58  59  60]
 [ 61  62  63  64  65  66  67  68  69  70]
 [ 71  72  73  74  75  76  77  78  79  80]
 [ 81  82  83  84  85  86  87  88  89  90]
 [ 91  92  93  94  95  96  97  98  99 100]]

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

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

[24 30 36 24 30 36]
[12 30 48 12 30 48]
```

¿Se pueden crear arreglos n-dimensionales? Si...

In [19]:

```
cubo = np.array([[[1], [2]], [[3], [4]]])
print(cubo)
print("Numero de elementos:", cubo.size)
print("Dimensiones:",np.ndim(cubo))
print("Forma del arreglo:", cubo.shape)
```

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

 [[3]
  [4]]]
Numero de elementos: 4
Dimensiones: 3
Forma del arreglo: (2, 2, 1)
```

Existen arreglos especiales que son de uso práctico en Algebra Lineal, por ejemplo:

In [3]:

```
# Arreglo de 1s
a = np.ones((3, 3))
print("np.ones((3, 3)) =")
print(a)
print()
# Arreglo de 0s
a = np.zeros((3, 3))
print("np.zeros((3, 3)) =")
print(a)
print()
# Arreglo Identidad
a = np.eye(5)
print("np.eye(5)=")
print(a)
print()
# Arreglo con una diagonal definida
a = np.diag(([1, 2, 3, 4, 5]))
print("np.diag([1, 2, 3, 4, 5]) =")
print(a)
print()
```

```
np.ones((3, 3)) =
[[ 1.  1.  1.]
 [ 1.  1.  1.]
 [ 1.  1.  1.]]

np.zeros((3, 3)) =
[[ 0.  0.  0.]
 [ 0.  0.  0.]
 [ 0.  0.  0.]]

np.eye(5)=
[[ 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.]]

np.diag([1, 2, 3, 4, 5]) =
[[1 0 0 0 0]
 [0 2 0 0 0]
 [0 0 3 0 0]
 [0 0 0 4 0]
 [0 0 0 0 5]]
```

Revisemos con mas detalle el tema de la seleccion de elementos respecto a sus indices (slicing) ya que en una matriz se tienen dos dimensiones

In [22]:

```
mat = np.arange(36).reshape((6, 6))    #np.reshape(array, (m, n)) -> array(m, n)
print("mat=\n",mat); print()
print("mat[0,0] =", mat[0,0]); print()
print("mat[2,2] =", mat[2,2]); print()
print("mat[-1,-1] =", mat[-1,-1]); print()
print("mat[:,1] =", mat[:,1]); print()
print("mat[2,:] =", mat[2,:]); print()
print("mat[:,1:2] =\n", mat[:,1:2]); print()
print("mat[:,1:3] =\n", mat[:,1:3]); print()
```

```
mat=
 [[ 0  1  2  3  4  5]
 [ 6  7  8  9 10 11]
 [12 13 14 15 16 17]
 [18 19 20 21 22 23]
 [24 25 26 27 28 29]
 [30 31 32 33 34 35]]

mat[0,0] = 0

mat[2,2] = 14

mat[-1,-1] = 35

mat[:,1] = [ 1  7 13 19 25 31]

mat[2,:] = [12 13 14 15 16 17]

mat[:,1:2] =
 [[ 1]
 [ 7]
 [13]
 [19]
 [25]
 [31]]

mat[:,1:3] =
 [[ 1  2]
 [ 7  8]
 [13 14]
 [19 20]
 [25 26]
 [31 32]]
```

Utilizando slicing se pueden modificar los elementos de un arreglo

In [23]:

```
mat = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print(mat,"\n")
# Se asigna 10 al elemento elemento en [0,0]
mat[0, 0] = 10
print(mat,"\n")
# Se asigna 0 a toda la segunda columna
mat[:, 1] = 0
print(mat)
```

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

[[10  2  3]
 [ 4  5  6]
 [ 7  8  9]]

[[10  0  3]
 [ 4  0  6]
 [ 7  0  9]]
```

Numeros aleatorios en numpy

numpy incluye un generador de numeros aleatorios que retorna arreglos

In [24]:

```
a = np.random.rand(4)     # Retorna 4 valores uniformes [0.0 - 1.0>
print(a)
a = np.random.randn(4)    # Distribucion normal (gaussiana)
print(a)
a = np.random.randint(0, 10, 5)  # Retorna 5 valores enteros ente [0 - 9]
print(a)
a = np.random.uniform(0, 10, 5)  # Retorna 5 valores reales entre [0.0 - 9.9]
print(a)
a = np.random.random((1, 2))    # Retorna un arreglo de m,n con valores reales entre [0.0 - 1.0>
print(a)
#Formula para [a-b> : (b - a) * np.random.random(m,n) + a
```

```
[ 0.92897951  0.77504584  0.91148797  0.43868842]
[-1.58781906 -0.04658426 -0.51612611  0.5437816 ]
[2 6 9 2 9]
[ 3.29693893  9.90437167  7.89236823  9.56107082  8.86806133]
[[ 0.59957319  0.93020355]]
```

Operaciones matemáticas en numpy

numpy incluye muchas de las funciones matematicas dispoibles en math, pero para ser aplicadas directamente sobre un arreglo

In [25]:

```
ang = np.linspace(-2*np.pi, 2*np.pi, 20)
y = np.sin(ang)
print("sin(x) =",y)
# Valores de exp(x) para x [-1, 1]
x = np.linspace(-1, 1, 10)
y = np.exp(x)
print("\nexp(x)=",y)
# Valores de Ln(x) entre [1 - 10]
x = np.arange(1, 10, 0.1)
y = np.log(x + 10e-20)       # Para evitar la division entre 0!!!
print("\nln(x)=", y)
```

```
sin(x) = [  2.44929360e-16   6.14212713e-01   9.69400266e-01   9.15773327e-01
   4.75947393e-01  -1.64594590e-01  -7.35723911e-01  -9.96584493e-01
  -8.37166478e-01  -3.24699469e-01   3.24699469e-01   8.37166478e-01
   9.96584493e-01   7.35723911e-01   1.64594590e-01  -4.75947393e-01
  -9.15773327e-01  -9.69400266e-01  -6.14212713e-01  -2.44929360e-16]

exp(x)= [ 0.36787944  0.45942582  0.57375342  0.71653131  0.89483932  1.11751907
  1.39561243  1.742909    2.17662993  2.71828183]

ln(x)= [ 0.          0.09531018  0.18232156  0.26236426  0.33647224  0.40546511
  0.47000363  0.53062825  0.58778666  0.64185389  0.69314718  0.74193734
  0.78845736  0.83290912  0.87546874  0.91629073  0.95551145  0.99325177
  1.02961942  1.06471074  1.09861229  1.13140211  1.16315081  1.19392247
  1.22377543  1.25276297  1.28093385  1.30833282  1.33500107  1.36097655
  1.38629436  1.41098697  1.43508453  1.45861502  1.48160454  1.5040774
  1.5260563   1.54756251  1.56861592  1.58923521  1.60943791  1.62924054
  1.64865863  1.66770682  1.68639895  1.70474809  1.7227666   1.74046617
  1.75785792  1.77495235  1.79175947  1.80828877  1.82454929  1.84054963
  1.85629799  1.87180218  1.88706965  1.90210753  1.91692261  1.93152141
  1.94591015  1.96009478  1.97408103  1.98787435  2.00148     2.01490302
  2.02814825  2.04122033  2.05412373  2.06686276  2.07944154  2.09186406
  2.10413415  2.11625551  2.12823171  2.14006616  2.1517622   2.16332303
  2.17475172  2.18605128  2.19722458  2.20827441  2.21920348  2.2300144
  2.24070969  2.2512918   2.2617631   2.27212589  2.28238239  2.29253476]
```

Asi también hay ciertas operaciones que requieren especificar el "eje" que afectara la operacion. Por ejemplo el método *np.sum, np.prod, np.cumsum, np.cumprod*, etc. El eje 0 es el eje de las columnas y el eje 1 es el de las filas. Se muestra un ejemplo con *np.sum*:

In [47]:

```
vec = np.array([1, 2, 3, 4, 5])
print("vec =",vec)
print("sum(vec) =",np.sum(vec))
print()

mat = np.array([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
print("mat =\n", mat)
print("sum(mat) = ", np.sum(mat))
print("sum(mat, axis = 0) = ", np.sum(mat, axis=0))
print("sum(mat, axis = 1) = ", np.sum(mat, axis=1))
```

```
vec = [1 2 3 4 5]
sum(vec) = [ 1  3  6 10 15]

mat =
 [[1 2 3]
 [4 5 6]
 [7 8 9]]
sum(mat) =  45
sum(mat, axis = 0) =  [12 15 18]
sum(mat, axis = 1) =  [ 6 15 24]
```

In [10]:

```
# Indexamiento booleano
A = np.random.randint(1, 100, 25)
print(A)
print()
print(A > 25)
print("Todos los numeros por encima de la media:")
print(A[A > np.mean(A)])
print("Todos los pares:")
print(A[A % 2 == 0])
print("Todos los multiplos de 7:")
print(A[A % 7 == 0])
```

```
[53 87 25 29 71 39 11 95 91 45 30  1 25 53 66 16 84 18 21 99 50 59 52 81
 94]

[ True  True False  True  True  True False  True  True  True  True False
 False  True  True False  True False False  True  True  True  True  True
  True]
Todos los numeros por encima de la media:
[53 87 71 95 91 53 66 84 99 59 52 81 94]
Todos los pares:
[30 66 16 84 18 50 52 94]
Todos los multiplos de 7:
[91 84 21]
```

In [11]:

```
#%%
# Crear una tabla de notas para 25 alumnos (notas aleatorias) y sacar los promedios
# asi como las estadisticas de la muestra
import numpy as np

notas = np.random.randint(5, 18, 4*25).reshape(25, 4)
#print(notas)

promedio = np.sum(notas, axis=1) / 4

# Mostrar los resultados de forma ordenada
for idx in range(25):
    print("Alumno {:2d} : \tPC1: {:2d}   PC2: {:2d}   PC3: {:2d}   PC4: {:2d}   PROM: {:4.1f}".\
          format(idx+1, notas[idx][0],notas[idx][1],notas[idx][2],notas[idx][3], promedio[idx]))

print("\nNumero de aprobados:", promedio[promedio >= 10.5].size)
```

```
Alumno  1 :     PC1: 10   PC2: 11   PC3: 10   PC4: 12   PROM: 10.8
Alumno  2 :     PC1: 13   PC2: 11   PC3:  8   PC4: 11   PROM: 10.8
Alumno  3 :     PC1:  8   PC2: 14   PC3: 11   PC4: 13   PROM: 11.5
Alumno  4 :     PC1: 17   PC2:  5   PC3:  5   PC4: 15   PROM: 10.5
Alumno  5 :     PC1: 11   PC2: 10   PC3:  8   PC4: 13   PROM: 10.5
Alumno  6 :     PC1: 12   PC2: 11   PC3: 13   PC4: 12   PROM: 12.0
Alumno  7 :     PC1: 15   PC2: 16   PC3: 17   PC4:  6   PROM: 13.5
Alumno  8 :     PC1: 12   PC2: 14   PC3: 11   PC4: 17   PROM: 13.5
Alumno  9 :     PC1: 13   PC2:  5   PC3:  6   PC4:  7   PROM:  7.8
Alumno 10 :     PC1: 17   PC2:  7   PC3: 15   PC4:  8   PROM: 11.8
Alumno 11 :     PC1: 14   PC2: 10   PC3: 11   PC4: 11   PROM: 11.5
Alumno 12 :     PC1: 16   PC2: 11   PC3: 11   PC4:  5   PROM: 10.8
Alumno 13 :     PC1:  6   PC2: 17   PC3: 12   PC4:  9   PROM: 11.0
Alumno 14 :     PC1: 14   PC2:  9   PC3: 10   PC4: 12   PROM: 11.2
Alumno 15 :     PC1: 12   PC2: 15   PC3: 11   PC4: 14   PROM: 13.0
Alumno 16 :     PC1:  8   PC2: 16   PC3: 16   PC4: 17   PROM: 14.2
Alumno 17 :     PC1:  9   PC2: 17   PC3:  8   PC4:  8   PROM: 10.5
Alumno 18 :     PC1:  7   PC2: 11   PC3: 16   PC4:  6   PROM: 10.0
Alumno 19 :     PC1: 12   PC2: 14   PC3: 13   PC4: 14   PROM: 13.2
Alumno 20 :     PC1: 15   PC2: 15   PC3:  8   PC4:  5   PROM: 10.8
Alumno 21 :     PC1:  8   PC2: 13   PC3: 10   PC4: 14   PROM: 11.2
Alumno 22 :     PC1: 10   PC2:  7   PC3: 17   PC4: 12   PROM: 11.5
Alumno 23 :     PC1:  8   PC2: 12   PC3:  5   PC4: 13   PROM:  9.5
Alumno 24 :     PC1:  6   PC2:  5   PC3: 15   PC4:  8   PROM:  8.5
Alumno 25 :     PC1: 16   PC2: 17   PC3: 15   PC4: 11   PROM: 14.8

Numero de aprobados: 21
```

Algebra Lineal con numpy

Los arreglos de 2 dimensiones Matrices son especiales en ciencias e ingenieria para la solucion de ciertos problemas de algebra lineal. Es por eso que existe la clase especial matrix:

In [48]:

```
# Arreglos
A_array = np.array([[5, 2,  4],[1, 7, -3],[6, -10, 0]])
B_array = np.array([[11, 5,  -3],[0, 7, -3],[6, -10, 0]])

# Matrices
A_matrix = np.matrix(A_array)
B_matrix = np.matrix(B_array)

#Se imprimen los tipos
print("A_array ->", type(A_array))
print("A_matrix ->", type(A_matrix))

# Se imprimen los productos de A * B
# Seran diferentes ya que la operacion * sera elemento-a-elemento en arreglo y producto-punto en matric
es
print("\nA_array * B_array =\n", A_array * B_array)
print("\nA_matrix * B_matrix =\n", A_matrix * B_matrix)
```

```
A_array -> <class 'numpy.ndarray'>
A_matrix -> <class 'numpy.matrixlib.defmatrix.matrix'>

A_array * B_array =
 [[ 55  10 -12]
 [  0  49   9]
 [ 36 100   0]]

A_matrix * B_matrix =
 [[ 79  -1 -21]
 [ -7  84 -24]
 [ 66 -40  12]]
```

En el caso de matrices hay algunos metodos (de la clase *"linalg"*) utiles:

In [49]:

```
# Otra forma de definir una matriz (estilo MATLAB)
A = np.matrix('1, 2, 3; 4, 5, 6; 7, 8, 9')
print("\nA =\n", A)
# La transpuesta es un atributo de una matriz
print("\nT(A) =\n", A.T)
# La inversa es un atributo de una matrix
print("\nInv(A) =\n", A.I)
# El determinante de una matriz es un metodo
print("\ndet(A) =\n", np.linalg.det(A))
```

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

T(A) =
 [[1 4 7]
 [2 5 8]
 [3 6 9]]

Inv(A) =
 [[ -4.50359963e+15   9.00719925e+15  -4.50359963e+15]
 [  9.00719925e+15  -1.80143985e+16   9.00719925e+15]
 [ -4.50359963e+15   9.00719925e+15  -4.50359963e+15]]

det(A) =
 6.66133814775e-16
```

Esto nos puede servir para resolver sistemas de ecuaciones simultaneas, operacion muy recurrente en ingeneria, utilizando la operacion matricial X = inv(A) * B

In [28]:

```
# 4x - 2y + 6z = 8
# 2x + 8y + 2z = 4
# 6x + 10y + 3z = 0
A = np.matrix('4, -2, 6; 2, 8, 2; 6, 10, 3')
B = np.matrix('8; 4; 0')
X = A.I * B
print("x =", float(X[0]))
print("y =", float(X[1]))
print("z =", float(X[2]))
```

```
x = -1.8048780487804876
y = 0.2926829268292683
z = 2.6341463414634143
```

EJEMPLOS DE APLICACION

EJEMPLO 1

Un proyectil se dispara con una velocidad de 750 m/s. Calcule la altura h que alcanza el proyectil si el angulo de lanzamiento Theta cambia de 5° a 85° en incrementos de 5°.

In [4]:

```
import numpy as np
# Variables inciales
vel = 750; g = 9.81
# Se genera el vector de angulos de lanzamiento
ang = np.arange(0, 86, 5)
# Se calcula el tiempo de vuelo para cada angulo
hmax = vel**2 * np.sin(2*np.radians(ang)) / g
# Se muestran los resultados
for a, h in zip(ang, hmax):
    print("angulo = {:5.2f}°    alcance = {:8.2f}m".format(a, h))
```

```
angulo =  0.00°    alcance =     0.00m
angulo =  5.00°    alcance =  9956.89m
angulo = 10.00°    alcance = 19611.25m
angulo = 15.00°    alcance = 28669.72m
angulo = 20.00°    alcance = 36857.09m
angulo = 25.00°    alcance = 43924.57m
angulo = 30.00°    alcance = 49657.42m
angulo = 35.00°    alcance = 53881.46m
angulo = 40.00°    alcance = 56468.33m
angulo = 45.00°    alcance = 57339.45m
angulo = 50.00°    alcance = 56468.33m
angulo = 55.00°    alcance = 53881.46m
angulo = 60.00°    alcance = 49657.42m
angulo = 65.00°    alcance = 43924.57m
angulo = 70.00°    alcance = 36857.09m
angulo = 75.00°    alcance = 28669.72m
angulo = 80.00°    alcance = 19611.25m
angulo = 85.00°    alcance =  9956.89m
```

EJEMPLO 2

Para calcular el coeficiente de rozamiento se mide la fuerza necesaria para mover una masa en reposo segun la siguiente configuracion:

Se tienen los siguientes datos:

Experimento	1	2	3	4	5	6
Masa (kg)	2	4	5	10	20	50
Fuerza (N)	12.5	23.5	30	61	118	294

Cual sera el valor de mu experimental?

In [30]:

```
import numpy as np

g = 9.81
m = np.array([2, 4, 5, 10, 20, 50])
F = np.array([12.5, 23.5, 30, 61, 118, 294])
mu = F / (m*g)
mu_media = np.mean(mu)
print("mu =", mu_media)
```

```
mu = 0.61170574244
```

EJEMPLO 3

Una empresa fabrica botellas de jugos por galón con tres tipos de sabores combinados: naranja, piña y mango. Las combinaciones tienen las siguientes combinaciones:

- 1 galon de mezcla uno: 3 cuartos de naranja, 0.75 cuartos de piña y 0.25 quartos de mango
- 1 galon de mezcla dos: 1 cuarto de naranja, 2.5 cuartos de piña y 0.5 quartos de mango
- 1 galon de mezcla tres: 0.5 cuartos de naranja, 0.5 cuartos de piña y 3 quartos de mango

¿Cuántos galones de cada mezcla pueden ser fabricados si 7,600 galones de jugo de naranja, 4,900 galones de piña y 3,500 galones de mango estan disponibles?

In [61]:

```
# Sistema de ecuaciones
# =====================
# x = galones de mezcla 1
# y = galones de mezcla 2
# z = galones de mezcla 3
#
# (3/4)x    + (1/4)y    + (0.5/4)z  = 7600
# (0.75/4)x + (2.5/4)y  + (0.5/4)z  = 4900
# (0.25/4)x + (0.5/4)y  + (3/4)z    = 3500

import numpy as np

A = np.matrix('3, 1, 0.5; 0.75, 2.5, 0.5; 0.25, 0.5, 3') / 4
B = np.matrix('7600; 4900; 3500');
X = A.I * B

print("Galones de mezcla 1: {:.1f}".format(float(X[0])))
print("Galones de mezcla 2: {:.1f}".format(float(X[1])))
print("Galones de mezcla 3: {:.1f}".format(float(X[2])))
```

```
Galones de mezcla 1: 8000.0
Galones de mezcla 2: 4800.0
Galones de mezcla 3: 3200.0
```

In [ ]: