Chap05

July 31, 2020

1 Capítulo 5

1.1 Arreglos: Vectores y Matrices

Hasta el momento hemos trabajado con variables que representan un sólo número (o conjunto de letras), pero en muchos casos se hace necesario trabajar con una lista de valores (un vector) o arreglos en 2D o 3D. Multiplicación de vectores o matrices pueden ser útiles en métodos computacionales, y Python permite realizar estas operaciones.

En Python hay varios tipos de estructuras de datos. Entre las más comunes están las *list*, *tuples* y *dictionaries*. De manera muy sencilla estos se describen como: - **list**

como su nombre lo indica, son una lista de valores. Cada valor está numerado empezando en cero – el primer valor está en la posición 0, el segundo en la posición 1, etc. Uno puede remover valores de una lista, adicionar valores a la lista, etc. Adicionalmente, los valores dentro de una lista pueden ser de diferente tipo, por ejemplo números y palabras. - **tuples**

son similares a las listas, pero no se puede cambiar su valor. Los valores que se le ponen a los tuples no pueden ser cambiados dentro del programa. La númeración es igual a la de las listas empezando en cero. Un posible uso es los nombres de los meses del año, que no cambiarán.

- **dictionaries**

como su nombre lo indica, son un diccionario. En un diccionario se tiene un índice de palabras, y para cada nombre una definición. En Python, la palabra se conoce como key y la definición el value. Los valores del diccionario no están numerados y no están ordenados en ningún orden específico. Se puede adicionar, quitar o modificar los valores del diccionario. Un ejemplo, es un directorio telefónico.

En este curso, el objetivo es realizaer trabajo numérico sobre valores de algún tipo, y por esta razón el uso de estas estructuras no es la más adecuada (esto incluye sumar valores a un vector, realizar multiplicación de matrices, rotación, etc). Por eso vamos a utilizar arreglos (numéricos o de otro tipo). Los arreglos son como listas, pero sólo aceptan un tipo de entrada (números en la mayoría de los casos). Para la creación y manejo de estos arreglos, vamos a utilizar los módulos de NumPy, por lo que siempre vamos a importar sus funciones a través de:

```
import numpy as np
```

y con esto podemos crear arreglos numéricos de manera sencilla.

1.2 Arreglos Numéricos

Para iniciar, un ejemplo para crear arreglos, que en general se pueden pensar como vectores)

[10]:
```
import numpy as np
a = np.array([1, 2, 3, 4, 5])
print(a)
```

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

donde se define un arreglo a con 5 números del 1 al 5. Recuerde que en Python (como en C) el contador empieza en cero.

[11]:
```
a[0]
```

[11]: 1

[12]:
```
a[4]
```

[12]: 5

Note que para saber el valor del arreglo en alguna posición se usa `[]`.

1.2.1 Hazlo tu mismo

Números primos Este es un ejercicio que nos permite pensar como se puede programar un problema numérico. El objetivo es determinar los números primos e imprimirlos. Sencillo cierto?

Qué es un número primo? un número natural mayor a 1 que tiene únicamente dos divisores positivos distintos: él mismo y el 1. En palabras sencillos, números enteros que no se puedan dividir por otro entero deferente de 1 y si mismo. Es decir

```
2, 3, 5, 7, 11, 13, ...
```

Un método muy sencillo que vamos a utilizar se conoce como la criba de Eratostenes (link), en honor al matemático griego del 3er siglo AC.

Se empieza con una lista de números del 2 al 100 y se considera que todos son primos, marcándolos con 0.

```
num  Primo?
 2     0
 3     0
 4     0
 5     0
 6     0
 7     0
 8     0
 9     0
 ...
 99    0
100    0
```

La estrategia es marcar todos los números que no son primos, poniendo el valor de 1 (indica que no es un primo). Cómo?

Se empieza con el número 2, (`num=2`) y se eliminan todos los múltiplos de 2 hasta el máximo que en nuestro caso es 100. El 2 es primo, ya que la segunda columna es cero.

```
num  Primo?
 2     0
 3     0
 4     1
 5     0
 6     1
 7     0
 8     1
 9     0
 ...
 99    0
100    1
```

Seguimos con el 3, un primo ya que la segunda columna sigue 0. (`num=3`) y se eliminan sus múltiplos.

```
num  Primo?
 2     0
 3     0
 4     1
 5     0
 6     1
 7     0
 8     1
 9     1
 ...
 99    1
100    1
```

El 4 lo saltamos, ya que el 4 y todos sus múltiplos son múltiplos del 2 y ya han sido eliminados. El 5 lo hacemos (es un primo) y continuamos sucesivamente.

Este es el concepto que tenemos que programar

Nota: Se debe evaluar todos los números `num`, hasta 10 (la raiz cuadrada de 100), porque los factores mas grandes ya han sido eliminados**.

Cuando terminamos, simplemente se imprime la posición el valor de `num` en el cual el `primo?` continue siendo cero. Podemos contar cuantos espacios tienen cero (total de primos entre 2 y 100) y además podemos saber qué números son primos (por su posición).

[13]:
```python
# prime.py
# Programa para encontrar números primos hasta el 100
import numpy as np

maxnum = 100
prime  = np.zeros(maxnum)

max1 = int(np.floor(np.sqrt(maxnum)))
```

```
for i in range(2,max1+1):
   if (prime[i-1]==0):                  # for i-1 = 2
      max2 = int(np.floor(maxnum/i))    # max2 = 50
      for j in range(2,max2+1):
         prime[i*j-1] = 1               # 4, 6, 8, (-1)

nprime = 0
for i in range(2,maxnum+1):
   if (prime[i-1]==0):
      nprime = nprime + 1
      print ("%4i" % i)

print ("# primos encontrados ", nprime)
```

```
   2
   3
   5
   7
  11
  13
  17
  19
  23
  29
  31
  37
  41
  43
  47
  53
  59
  61
  67
  71
  73
  79
  83
  89
  97
# primos encontrados  25
```

Explicación El programa comienza definiendo un arreglo `prime` lleno de ceros

```
maxnum = 100
prime  = np.zeros(maxnum)
```

Para eliminar números que no sean primos, usa dos loops, el primero

```
for i in range(2,max1+1):
```

que mira los posibles números 2, 3, 4, 5,...,10. Si el número no ha sido multiplo de ningún número anterior

`if (prime[i-1]==0):`

se realiza otro loop, en el que se multiplica ese número por 2, 3, 4, 5, etc. para eliminar todos sus posibles múltiplos.

```
for j in range(2,max2+1):
   prime[i*j-1] = 1
```

esto se hace con `i*j`. Recuerde que Python empieza el conteo con `i=0`, por eso se escribe `i*j-1`.

Las variables `max1` y `max2` se definen para no tener multiplos mayores a `maxnum=100`.

1.2.2 Hazlo tu mismo

Números primos V2.0 La versión anterior de la busqueda de números primos funciona bastante bien. **Sin embargo tiene una desventaja**. Si uno no quiere los primos hasta el 100, sino hasta el 1000? Tendría 168 primos (verifique con su código), e imprimiría muchas líneas y sería difícil de mirar el resultado.

Cree una función propia, que use el mismo código anterior, pero que como resultado devuelva un vector con los números enteros primos.

[14]:
```
def primos_vector(maxnum):
    """
    prime_vector(maxnum)
    Función que busca números primos entre 2 y maxnum,
    y los ubica en un arreglo. El programa es muy lento si
    se buscan primos muy grandes.
    Entradas:
       maxnum - entero, busqueda de primos hasta maxnum

    Salidas:
       pvec   - vector con números primos, en orden
       nprime - primos encontrados
    """
    import numpy as np

    prime    = np.zeros(maxnum,dtype=int)
    prime[0] = 1

    max1 = int(np.floor(np.sqrt(maxnum)))
    for i in range(2,max1+1):
        if (prime[i-1]==0):
            max2 = int(np.floor(maxnum/i))
            for j in range(2,max2+1):
                prime[i*j-1] = 1

    # número de primos, y crear vector
```

```
    nprime = np.count_nonzero(prime==0)
    pvec   = np.zeros(nprime,dtype=int)
    pcnt = 0
    for i in range(2,maxnum+1):
        if (prime[i-1]==0):
            pcnt = pcnt + 1
            pvec[pcnt-1] = i

    return pvec,nprime
```

[15]:
```
# prime2.py

primes,nprime = primos_vector(1000)

print ("# primos encpontrados ", nprime)
print (primes)
```

```
# primos encpontrados  168
[  2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61
  67  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151
 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251
 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359
 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463
 467 479 487 491 499 503 509 521 523 541 547 557 563 569 571 577 587 593
 599 601 607 613 617 619 631 641 643 647 653 659 661 673 677 683 691 701
 709 719 727 733 739 743 751 757 761 769 773 787 797 809 811 821 823 827
 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937 941 947 953
 967 971 977 983 991 997]
```

La ventaje de imprimir el arreglo `primes` es que Python automáticamente despliega el arreglo con varias filas y columnas. Así se puede ver de manera sencilla los números primos.

Intente desplegar el resultado aumentando el número máximo (después de 1 millón, el programa puede ser muy lento).

1.3 Uso de arreglos

Los valores de un arreglo se pueden asignar uno a la vez o todos a la vez con comandos sencillos, como en el siguiente programa:

[16]:
```
# testarreglos.py
# Programa que muestra como generar arreglos
#
import numpy as np

x = np.array([1, 2, 3])
y = np.ones(3)
z = np.ones(3)*2
```

```
# Imprima los tres arreglos
print ('x = ',x)
print ('y = ',y)
print ('z = ',z)

# Asigne algunos valores
x[1] = 15
y[:] = 2
z = z-1

# Imprima otra vez arreglos
print('')
print('x = ',x)
print('y = ',y)
print('z = ',z)
```

```
x =  [1 2 3]
y =  [1. 1. 1.]
z =  [2. 2. 2.]

x =  [ 1 15  3]
y =  [2. 2. 2.]
z =  [1. 1. 1.]
```

Explicación: Note que cuando un arreglo se le asigna un solo valor, cada elemento del arreglo toma ese valor(`y[:] = 2`), pero tenga cuidado ya que si Ud ejecuta `y = 2` el resultado no es un arreglo, es sólo una variable. Ud habrá reemplazado el arreglo con una variable sencilla `y`.

Para cambiar el valor de un elemento, se utiliza x[1]=15, que cambia el valor en la posición `1` del arreglo (es decir, el segundo número, recuerde Python empieza con cero). Tenga en cuenta que la posición dentro del vector o matriz, debe coincidir con el tamaño del arreglo. Si se usa una posición mayor a la disponible, Python genera un error.

1.3.1 Hazlo tu mismo

El programa `prime2.py` muestra el resultados de números primos hasta el 1000. Pero que hacemos si queremos imprimir los números en 10 columnas.

Busque una mejor manera de imprimir el resultado.

[17]:
```
# prime3.py

primes,nprime = primos_vector(1000)

print ("# primos encpontrados ", nprime)

nprint = 0
for i in range(1,nprime):
      nprint = nprint + 1
```

```
      if (nprint%10==0 ):
         print ("%4i" % (primes[i]))
      else:
         print ("%4i" % (primes[i]),end="")
print('')
```

```
# primos encpontrados  168
   3   5   7  11  13  17  19  23  29  31
  37  41  43  47  53  59  61  67  71  73
  79  83  89  97 101 103 107 109 113 127
 131 137 139 149 151 157 163 167 173 179
 181 191 193 197 199 211 223 227 229 233
 239 241 251 257 263 269 271 277 281 283
 293 307 311 313 317 331 337 347 349 353
 359 367 373 379 383 389 397 401 409 419
 421 431 433 439 443 449 457 461 463 467
 479 487 491 499 503 509 521 523 541 547
 557 563 569 571 577 587 593 599 601 607
 613 617 619 631 641 643 647 653 659 661
 673 677 683 691 701 709 719 727 733 739
 743 751 757 761 769 773 787 797 809 811
 821 823 827 829 839 853 857 859 863 877
 881 883 887 907 911 919 929 937 941 947
 953 967 971 977 983 991 997
```

El comando

```
print ("%4i" % (primes[i]),end="")
```

hace lo mismo que el comando `print` pero no genera un salto de linea. El código simplemente revisa si ya se han imprimido 10 números primos (con el contador `nprint`), y permite que se salta la línea.

1.4 Arreglos 2D

Arreglos en 2 o mas dimensiones se pueden generar en Python utilizando `NumPy`. Abajo un programa que muestra algunas características de arreglos en 2D:

[30]:
```
# testmatriz.py
# programa con algunos ejemplos de matrices 2D
#
import numpy as np

y = np.empty([2, 3])
print('Empty Y float ')
print(y)

x = np.zeros([2,3],dtype=int)
print('Zeros X enteros')
```

```
print(x)

x[0,:] = [1, 2, 3]
x[1,:] = [4, 5, 6]
print('X unidades')
print (x)

x = x + 1
print('X mas 1')
print(x)

c = np.array( [ [1,2], [3,4] ], dtype=complex )
print('Matriz compleja')
print(c)
```

```
Empty Y
[[4.9e-324 9.9e-324 1.5e-323]
 [2.0e-323 2.5e-323 3.0e-323]]
Zeros X
[[0 0 0]
 [0 0 0]]
X unidades
[[1 2 3]
 [4 5 6]]
X mas 1
[[2 3 4]
 [5 6 7]]
Matriz compleja
[[1.+0.j 2.+0.j]
 [3.+0.j 4.+0.j]]
```

1.5 Aritmética con Arreglos

La ventaja de utilizar los arreglos con `NumPy` es que adicional a guardar los datos en un formato de arreglos, se pueden realizar operaciones matriciales con ellos. Python permite realizar operaciones en vectores y matrices utilizando comandos propios sin la necesidad de realizar la aritmética uno mismo. Abajo un ejemplo sencillo de algunas operaciones con vectores.

[43]:
```
# vectormath.py
# Program showing arithmetic operations with arrays
# in Python

import numpy as np
#import math as mt

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

```
print("a     = ", a)

c = a + 1
print("a + 1 = ", c)

c = 2 * a
print("2 * a = ", c)

c = a * a
print("a * a = ", c)

print('')
c = np.sqrt(a)
print("sqrt(a) = ", c)

c = np.sin(a)
print("sin(a) = ", c)

c = np.exp(a)
print("exp(a) = ", c)

print('')
print("a = ", a)
print("b = ", b)

c = a + b
print("a + b = ", c)

c = a * b
print("a * b = ", c)

print('')
x = np.sum(a)
print("sum(a) = ", x)

c = a
c[3:5] = 0.0
print ("a con dos ceros al final", c )

x = np.dot(a,b)
print ("a dot b = ", x)

print('')
x = np.sum((a - np.sum(a)/5.)**2)
print ("suma del cuadrado de las diferencias del promedio = ", x)
```

```
a     =  [1. 2. 3. 4. 5.]
a + 1 =  [2. 3. 4. 5. 6.]
2 * a =  [ 2.  4.  6.  8. 10.]
a * a =  [ 1.  4.  9. 16. 25.]

sqrt(a) =  [1.         1.41421356 1.73205081 2.         2.23606798]
sin(a) =  [ 0.84147098  0.90929743  0.14112001 -0.7568025  -0.95892427]
exp(a) =  [  2.71828183   7.3890561   20.08553692  54.59815003 148.4131591 ]

a =  [1. 2. 3. 4. 5.]
b =  [2. 2. 2. 2. 2.]
a + b =  [3. 4. 5. 6. 7.]
a * b =  [ 2.  4.  6.  8. 10.]

sum(a) =  15.0
a con dos ceros al final [1. 2. 3. 0. 0.]
a dot b =  12.0

suma del cuadrado de las diferencias del promedio =  6.800000000000001
```

Operaciones en matrices también están disponibles en {np}. Abajo algunos ejemplos:

[51]:
```
# matrixmath.py
# Operaciones matriciales en Python
#
import numpy as np

a = np.array( [[ -5.1, 3.8, 4.2 ], \
               [ 9.7, 1.3, -1.3]] )

b = np.empty( [3,2])
b[:, 0] = [9.4, -6.2, 0.5 ]
b[:, 1] = [-5.1, 3.3, -2.2]

print("Matrix a")
print(a)
print("Matrix b")
print(b)

c = np.matmul(a, b)
print ("matmul(a,b)")
print (c)

c = np.matmul(b,a)
print ("matmul(b,a)")
print (c)

print("Valor max de a ", np.amax(a))
```

```
loc2 = np.argmax(a)
loc1 = np.unravel_index(loc2, np.shape(a))
print("Posición del max of a", loc1)
print("Max(a) con su posición", a[loc1])

c = a + np.transpose(b)
print("Matrix a + transpose(b)")
print(c)

print("a shape ", np.shape(a))
print("b shape ", np.shape(b))

print("a size  ", np.size(a))
print("b size  ", np.size(b))
```

```
Matrix a
[[-5.1  3.8  4.2]
 [ 9.7  1.3 -1.3]]
Matrix b
[[ 9.4 -5.1]
 [-6.2  3.3]
 [ 0.5 -2.2]]
matmul(a,b)
[[-69.4   29.31]
 [ 82.47 -42.32]]
matmul(b,a)
[[-97.41  29.09  46.11]
 [ 63.63 -19.27 -30.33]
 [-23.89  -0.96   4.96]]
Valor max de a  9.7
Posición del max of a (1, 0)
Max(a) con su posición 9.7
Matrix a + transpose(b)
[[ 4.3 -2.4  4.7]
 [ 4.6  4.6 -3.5]]
a shape  (2, 3)
b shape  (3, 2)
a size   6
b size   6
```

Arriba se demuestra las función de {np} como {matmul}, {amax} y {argmax}, la transpuesta de una matriz y su tamaño. Note que {matmul(a,b)} es una multiplicación de matrices, no la multiplicación de los elementos de la matriz como resultado de {a*b}.

Es importante notar que multiplicar {matmul(a,b)} es correcto dado que las dimensiones son {(2,3)} y {(3,2)}. El caso contrario, tambien es posible. Si las dimensiones de las matrices no lo permiten, Python genera un error.

La función {np.argmax} reporte el índice del valores máximo dentro de una matriz (reporte el primer valor máximo), pero no reporta la posición del elemento en 2D, sólo la posición dentro de la matriz de acuerdo al orden de lectura. Para obtener los dos parámetros (posición en fila y columna), se utiliza {loc1 = np.unravel_index(loc2, np.shape(a))}.

Como es de esperarse, hay funciones para {amin} y {argmin} para determinar el valor mínimo de la matriz y su posición. La página de referencia para todas las funciones en {NumPy} se encuentra en el link.

[52]:
```
len(np.shape(a))
```

[52]: 2

[56]:
```
s = np.zeros([5,1])
print(np.shape(s))
print(len(np.shape(s)))
```

```
(5, 1)
2
```

1.6 Arreglos en funciones

Como ya se pudo corroborar, en la función primos_vector la función puede dar como resultado un arreglo. También puede recibir uno y hacer algo con él.

Suponga que Ud. quiere analizar los datos dentro de un arreglo

x = np.array([1 , 2, ..., 100])

con 100 elementos. Para mejorar la legibilidad de su código, esto lo quiere hacer dentro de una función en Python

x2,xsum = analisis(x)

donde se envía el vector `x', y se obtiene de vuelta otro vector `x2' y otras variables `xsum'. En este caso, todos los valores del arreglo estarán disponibles dentro de la función.

1.6.1 Hazlo tu mismo

Genere una función, que recibe un arreglo (una sola dimensión), y devuelve el vector pero con el promedio corregido, el promedio original y la varianza.

[82]:
```
def arr_trab(x):
    """
    Función para análisis de unos datos en vector
    Input: x     = array of given size, with numbers
           n     = size of the array
    Output y     = demeaned array
           x_mu  = valor promedio de x
           y_var = variance of array
```

```
    """
    import numpy as np

    n     = np.size(x)
    x_mu  = np.mean(x)
    y     = x-x_mu
    y_var = np.var(y)

    return y,x_mu,y_var
```

Evalue si su función está haciendo el trabajo bien.

[87]:
```
# array2fun.py
import numpy as np

mu    = 25
sigma = 3.0
a = np.random.normal(mu,sigma,5)
n = np.size(a)
print('Arreglo original ')
print (a)

[b,x_mean,x_var] = arr_trab(a)

print('Arreglo corregido (demeaned)')
print(b)

print("Prom(x) = ",x_mean)
print("Var(x)  = ",x_var)
```

```
Arreglo original
[19.74788005 26.44167912 24.13886532 26.85581016 23.78583781]
Arreglo corregido (demeaned)
[-4.44613444  2.24766463 -0.05514917  2.66179566 -0.40817668]
Prom(x) =  24.194014492914697
Var(x)  =  6.414982704558334
```

1.7 Arreglos de caracteres

Las cadenas de caracteres (strings) son uno de los tipos más comunes en Python. Se pueden crear simplemente al encerrarlos en comillas (dobles o sencillas). Como ya lo vimos al comienzo, una variable puede ser asigada un string. Note que todo lo que esté entre comillas va a pertenecer al string, incluidos los espacios en blanco. Para determinar el número de caracteres en un string

[100]:
```
a = "Hello World!"
b = "Python"
print(len(a))
```

```
print(len(b))
```

```
12
6
```

Note que el número de caracteres incluye los espacios en blanco, sean en la mitad, al comienzo o al final.

También es posible solicitar substrings, y crear nuevas variables concatenandolos.

En Python, parece que asignar caracteres a una subparte de una variable no es posible. Esto no se puede

```
a[0:5] = 'Hollo'
>>> TypeError: 'str' object does not support item assignment
```

Algunas operaciones con strings

[101]:
```
# char_operations.py
# Operaciones de string de caracteres
#

a = "Hello World!"
b = "Python"

c = a[0:6] + b # concatenation
print(c)

c = b*2 # Repetition
print(c)

print(a[:5],a[6:]) # Range Slice

# Find characters
i = a.find("ello")
print (i)

# Remove blanks
c = "   "+a+b+"  "
print(c)
d = c.strip()
print(d)

# Split string, with whitespace delimiter
c = a.split()
print(c)
```

```
Hello Python
PythonPython
```

```
Hello World!
1
   Hello World!Python
Hello World!Python
['Hello', 'World!']
```

1.7.1 Hazlo tu mismo

Preguntar al usuario, que jugador de futbol es el mejor del mundo. Dependiendo de la respuesta, el programa debe comentar algo obvio.

[103]:
```
# futbol.py
# Programa para interactuar con strings

a = input("Who is the best soccer player in the world ")

if (a.find("onal")>-1 or a.find("rist")>-1):
   print("Eres hincha del Real Madrid")
elif (a.find("Leo")>-1 or a.find("essi")>-1):
   print("Te gusta el Barcelona")
else:
   print("De acuerdo, James debe jugar mas en el RM")
```

```
Who is the best soccer player in the world Leo
Te gusta el Barcelona
```

Note que acá, se espera un conjunto de respuestas (Cristiano y Messi) para lo cual se responde fácilmente, mientras que si el usuario pone una respuesta distinta, sólo se responde de una manera, no importa quién sea el jugador que puso el usuario.

1.8 Arreglos de caracteres

Como se mostró arriba, un string es simplemente una cadena de caracteres. Por lo tanto, un arreglo 1D de strings sería en realidad un arreglo 2D. A diferencia de lo que se puede hacer en Fortran, en Python no es fácil crear arreglos de caracteres, y en cambio se usan listas list.

Un ejemplo del uso de listas en Python:

[104]:
```
# lista_caracter.py
# Uso de listas de strings de caracteres

lista = ['Ivan Duque', 'Juan M. Santos', 'Alvaro Uribe ']
print (len(lista))

print("Nuestro antiguo presidente fue ", lista[1])
print("Nuestro presidente actual es   ", lista[0])

lista.append("José Miguel Pey")
```

```
print("Y nuestro 1er presidente fue   ", lista[3])
```

```
3
Nuestro antiguo presidente fue  Juan M. Santos
Nuestro presidente actual es    Ivan Duque
Y nuestro 1er presidente fue    José Miguel Pey
```

En Python se pueden utilizar algunos métodos sobre las listas (sean de caracteres o números o ambos).

```
list.append(elem) -- adiciona un elemento a la lista
list.insert(index, elem) -- inserta un elemento, corre los
                            demás a la derecha
list.extend(list2) -- pega list2 a list. Se puede usar + o +=
list.index(elem) -- busqueda y posición de un elemento. Error si no existe.
list.remove(elem)
list.sort()
list.reverse()
list.pop(index)
```