Capítulo 5

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 utilixzr 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.

5.1. Arreglos Numéricos

Los arreglos numéricos en Python los vamos a hacer con los módulos de `NumPy`, un paquete para análisis numérico.

```
>>> import numpy as np
>>> a = np.array([1, 2, 3, 4, 5])
>>> array([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. Es decir

```
>>> a[0]
>>> 1
>>> a[4]
>>> 5
```

Para empezar, este es un ejemplo de un programa que usa arreglos para calcular números primos hasta el 100:

```
# prime.py
# Programa para calcular números primos hasta el 100

import numpy as np
import math as mt

maxnum = 100
prod = np.zeros(maxnum)

#maxnum=100, max_i = 10
max_i = mt.floor(mt.sqrt(float(maxnum)))

for i in range(2,max_i+1):
   if (prod[i-1]==0):   # for i-1 = 2
      max_j = round(maxnum/i)  # maxj = 50, maxj=33, etc.
      for j in range(2,max_j+1):
         prod[i*j-1] = 1    # 4, 6, 8, (-1)

nprime = 0
```

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

print ("Number of primes found ", nprime)
```

El método utilizado se conoce como la *criba de Eratostenes* (vea la animación en Wikipedia), 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. En el programa, esta lista se creo en el arreglo `prod`, que se inicializa con valores de cero en todos los puntos. La estrategia es *marcar* todos los números que no son primos, poniebdo el valor de 1, esto es `prod[i*j-1]`. Note que como Python empieza con el cero, toca siempre quitarle un punto a la ubicación dentro del vector o arreglo.

Se empieza con el número 2 (`i=2`) y se eliminan todos los múltiplos de 2 hasta el máximo que en nuestro caso es 100. Seguimos con el 3 `i=3`) y sus múltiplos. El 4 lo saltamos, ya que el 4 y todos sus múltiplos son múltiplos del 2 y ya han sido eliminados. Se debe evaluar todos los números `i`, 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 `i` en el cual el `prod[i-1]` continue siendo cero. Contamos la cantidad de números primos con el contador `nprime` e imprimimos el número de `i`.

Este programa imprime cada número primo en una línea, lo cual hace que el resultado sea muy largo si queremos más números primos. Podemos modificar el programa para que se imprima una matriz con 10 números por línea, poniendo los números primero en un vector largo, que luego se reorganize (`reshape`) para que tenga el tamaño ideal. El programa es el siguiente:

```
# prime2.py
# Program to calculate prime numbers
# A matrix with up to 200 prime numbers will we printed.

import numpy as np
import math as mt

maxnum = 2000

prod = np.zeros(maxnum)
pnum = np.zeros(200)

#maxnum=100, max_i = 10
max_i = mt.floor(mt.sqrt(float(maxnum)))

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

```
   if (prod[i-1]==0):   # for i-1 = 2
      max_j = mt.floor(maxnum/i)  # maxj = 50, maxj=33, etc.
      for j in range(2,max_j+1):
         prod[i*j-1] = 1    # 4, 6, 8, (-1)

nprime = 0

for i in range(2,maxnum+1):
   if (prod[i-1]==0):
      nprime = nprime + 1
      pnum[nprime-1] = i
   if (nprime==200):
      break

a = pnum.reshape(20, 10)
print (a)
```

El código no cambia, solo se adiciona

```
pnum[nprime-1] = i
```

para llenar el vector `pnum` con los números primos, y posteriormente se reorganiza

```
a = pnum.reshape(20, 10)
```

para tener una matriz con 20 filas y 10 columnas, que después se imprime en pantalla. La matriz `a` es un arreglo en dos dimensiones, que veremos con mayor detalle más adelante (sección 5.1.1).

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:

```
# testarray.py
# Program showing simple array handling

import numpy as np

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

x[1] = 15
y[:] = 2
z    = z-1
```

```
print(x)
print(y)
print(z)
```

con resultado

```
gprieto > python testarray.py
[1 2 3]
[ 1.  1.  1.]
[ 2.  2.  2.]
[ 1 15  3]
[ 2.  2.  2.]
[ 1.  1.  1.]
```

Note que cuando un arreglo se le asigna un solo valor, cada elemento del arreglo toma ese valor (por ejemplo uando hicimos `y[:] = 2`), pero tenga cuidad que si Ud. ejecuta `y=2` el vector `y` deja de ser un vector y se vuelve una variable sencilla.

Para cambiar el valor de un elemento, se utiliza `x[1]=15` para cambiar únicamente ese valor al elemento 1 del arreglo. 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.

En algunos casos, en vez de crear una matriz, se quiere imprimir un comando, pero no necesariamente avanzar una línea (como hacer un `enter`, de tal manera que se impriman varios comandos en la misma linea. El ejemplo de los números primos con el comando para imprimir sin avanzar y sin crear un arreglo:

```
# prime3.py
# Programa para calcular números primos hasta el 100

import numpy as np
import math as mt

maxnum = 1000

prod = np.zeros(maxnum)

#maxnum=100, max_i = 10
max_i = mt.floor(mt.sqrt(float(maxnum)))

for i in range(2,max_i+1):
   if (prod[i-1]==0):   # for i-1 = 2
      max_j = mt.floor(maxnum/i)  # maxj = 50, maxj=33, etc.
      for j in range(2,max_j+1):
         prod[i*j-1] = 1    # 4, 6, 8, (-1)
```

```
nprime = 0
for i in range(2,maxnum+1):
   if (prod[i-1]==0):
      nprime = nprime + 1
      if (nprime%10==0 ):
         print ("%4i" % (i))
      else:
         print ("%4i" % (i),end="")

print ("")
print ("Number of primes found ", nprime)
```

El comando

```
print ("%4i" % (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 `nprime`, y permite que se salta la línea.

5.1.1. Arreglos en 2D

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

```
# testmatrix.py
# Program showing simple matrix array handling

import numpy as np

x = np.empty((2, 3))
print (x)

x[0,:] = [1, 2, 3]
x[1,:] = [4, 5, 6]

print (x)

x = x + 1
print(x)

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

que tiene como resultado

```
[[1152921504606846976 1152921504606846976          4546887684]
 [         4546925168          4546925168          4551296520]]
[[1 2 3]
 [4 5 6]]
[[2 3 4]
 [5 6 7]]
[[ 1.+0.j  2.+0.j]
 [ 3.+0.j  4.+0.j]]
```

Es importante tener en cuenta el orden como Python (y otros lenguajes) guardan la información de un arreglo, pues esto es lo que otras rutinas van a utilizar. Siempre se tiene `x(i,j)` donde `i` son las filas y `j` son las columnas.

5.1.2. Aritmética en 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.

```
# 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)

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("b = ", b)

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

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

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

c = a
c[3:4] = 0.0
print ("a with two zeros at end", c )

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

x = np.sum((a - np.sum(a)/5.)**2)
print ("sum of squares of difference from mean = ", x)
```

Note que operaciones complicadas son posibles con `NumPy`. Es importante saber si el resultado es un escalar `x` o una matriz `c`. Y tenga en cuenta que para operaciones como raiz cuadrada `sqrt()` se utiliza la función perteneciente a `np` y no la del módulo `math`, ya que la última sólo trabaja con escalares. El resultado del programa anterior:

```
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) =  [ 0.8414  0.9092  0.1411 -0.756  -0.9589]
exp(a) =  [   2.7182    7.389    20.0855   54.5981  148.413 ]
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 with two zeros at end [ 1.  2.  3.  0.  5.]
a dot b =  22.0
sum of squares of difference from mean =  14.8
```

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

```
# vectormath.py
# Program showing arithmetic operations with arrays
# in 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 ("BAD. Careful, matmul does not check array sizes")

print("Max value of a ", np.amax(a))
loc2 = np.argmax(a)
loc1 = np.unravel_index(loc2, np.shape(a))
print("Position of max of a", loc1)
print("Max(a) using location", 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))
```

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, `matmul(b,a)` no debería ser posible, sin embargo Python no parece caer en cuenta de esto y realiza la igual multiplicación, adicionando ceros a las matrices para ajustar sus tamaños (**tener mucho cuidado con esto**).

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 `https://docs.scipy.org/doc/numpy/reference/`.

5.2. Arreglos en funciones

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,n)
```

donde se manda el vector `x` y su tamaño `n`, y se obtiene de vuelta otro vector y otras variables. En este caso, todos los valores del arreglo estarán disponibles dentro de la función.

En el siguiente ejemplo se pasa un vector a una función, la cual calcula el promedio y la varianza, y genera un vector de salida con el promedio removido.

```
# array2fun.py
# Program showing working with arrays in functions

def arr_work(x,n):
   # Function to do some basic analysis of an array of size n
   # Input: x     = array of given size, with numbers
   #        n     = size of the array
   # Output y     = demeaned array
   #        y_var = variance of array
```

```
   import numpy as np

   if (np.size(x) != n):
      print("problem with size")
      return -1,-1,-1

   x_mean = np.mean(x)
   y = x-x_mean
   y_var = np.var(y)

   return [y,y_var]

#--------------------
# Main program

import numpy as np

a = np.random.rand(5)
n = np.size(a)

print (a)

[b,b_var] = arr_work(a,n)

print(b)
print(np.var(a),b_var)
```

Como resultado obtenemos

```
gprieto > python array2fun.py
[ 0.14033254  0.15790634  0.2586828   0.86901262  0.59010309]
[-0.26287494 -0.24530114 -0.14452468  0.46580515  0.18689561]
Mean(x) =  0.403207477864
Var(x)  =  0.0804135341731
gprieto > python array2fun.py
[ 0.13711914  0.82462952  0.49242041  0.27332684  0.91166646]
[-0.39071334  0.29679705 -0.03541206 -0.25450563  0.38383398]
Mean(x) =  0.527832475382
Var(x)  =  0.0908202118678
gprieto > python array2fun.py
[ 0.26828717  0.53819358  0.09357034  0.8474883   0.02242354]
[-0.08570542  0.18420099 -0.26042224  0.49349571 -0.33156904]
Mean(x) =  0.353992586126
Var(x)  =  0.0925142434748
```

Note que entradas y salidas pueden incluir variables y arreglos.

5.3. 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`:

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

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`

```
len(a)
len(b)
```

se obtiene como resultado `12` y `6` respectivamente. 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.

```
a[0]     = 'H'
a[0:5]   = 'Hello'
b[0:6]   = 'Python'
a[0:6]+b = 'Hello Python'
```

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
```

En el ejemplo de abajo, se muestran varias operaciones básicas con `strings` que pueden ser útiles, incluyendo concatenación, repetición, remoción de espacios en blanco, encontrar cadena de caracteres, etc.

```
# char_operations.py
# Basic character string operations

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)
```

Un ejemplo de cómo esto puede ser útil en programaci?øn, se muestra en el ejemplo del mejor jugador de fútbol

```
# 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")
```

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.

5.3.1. 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:

```
# lista_caracter.py
# Uso de listas de strings de caracteres
```

```
lista = ['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[2])
```

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)
```

Problemas

5.1. Basado en el programa `prime.py`, cree un programa que genere una matriz en la cual se organicen los números primos. Pero en este caso, organícelos en forma de hélice, es decir

```
2x2 matrix
[[ 2.  3.]
 [ 7.  5.]]
3x3 matrix
[[ 17.  19.  23.]
 [ 13.   2.   3.]
 [ 11.   7.   5.]]
4x4 matrix
[[ 17.  19.  23.  29.]
 [ 13.   2.   3.  31.]
 [ 11.   7.   5.  37.]
 [ 53.  47.  43.  41.]]
```

5.2. Un análisis simple de una serie de datos incluye el cálculo del promedio (mean) y la desviación estandard (std). Sin embargo, algunos datos tienen

distribuciones estadísticas que no son normales (gaussiana), y una mejor descripción del valor medio de los datos se obtiene con la mediana (median). Para calcular la mediana, se requiere organizar los datos de manera ascendente y así determinar la mediana. Una función que permite reorganizar un arreglo de números utiliza el método de `quicksort`, que se muestra abajo:

```
def quicksort(arr_in):
   # SImple routine for a quicksort
   # input: arr_in - array with numbers, unsorted
   #
   # output: arr - same array, but sorted
   #

   import numpy as np

   if (arr_in.ndim>1):
      print("Array is wrong dimensions")
      return None

   # Make a copy of the array
   arr = np.copy(arr_in)
   n   = arr.size

   for j in range(2,n+1):
      ibreak = 0
      A = arr[j-1]
      for i in range(j-1,0,-1):
         if (arr[i-1] <= A):
            arr[i] = A
            ibreak = 1
            break
         arr[i]=arr[i-1]
      if (ibreak==0):
         arr[0] = A
   return arr
```

Cree dos arreglos de 50 y 51 números aleatorios con `numpy` y con funciones determine el promedio, desviación estándar y la mediana del arreglo. No utilice funciones propias de Python para el cálculo. Compare sus resultados con las funciones `np.mean` y `np.median`.

Puede usar

```
C = np.random.standard_normal(51)
```

para crear su arreglo.