

ARREGLOS VECTORES Y MATRICES EN PYTHON

UNPAZ

Introducción a la Programación

Arreglos

Introducción

En programación y/o algoritmos, los arreglos (llamados en inglés arrays) son una zona de almacenamiento continuo, que incluye una serie de elementos del mismo tipo. Desde el punto de vista lógico un arreglo tipo vector se puede ver como un conjunto de elementos ordenados en fila.

Ejemplo:

Dimension miVector(3)			
miVector	2	3	4
laPosicion	1	2	3

Podemos concluir que un arreglo tiene:

- **Tamaño:** cuántas cajas va a tener, el número de datos.
- **Tipo:** cuál es el tipo de todos los datos del arreglo.
- **Índice o posición:** En algunos lenguajes el primer elemento de un arreglo tiene el índice o posición 0 .

Arreglos

Dimensiones de un Array

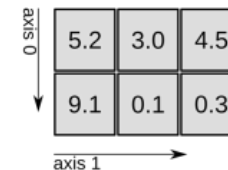
- El dimensionamiento de un arreglo definirá el tipo de estructura sobre la que trabajaremos. La misma está definida por la cantidad de columnas (podríamos llamarlo eje X) y la cantidad de filas (eje Y).
- Para una lista de valores se crea un array de una dimensión, también conocido como **vector**. **(1D)**
- Para una lista de listas de valores se crea un array de dos dimensiones, también conocido como **matriz** **(2D)**.
- Para una lista de listas de listas de valores se crea un array de tres dimensiones, también conocido como **cubo**. **(3D)**
- Y así sucesivamente. No hay límite en el número de dimensiones del array más allá de la memoria disponible en el sistema. (nD)

1D array



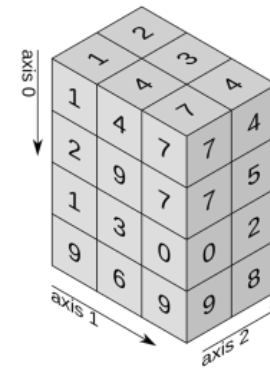
shape: (4,)

2D array



shape: (2, 3)

3D array



shape: (4, 3, 2)

Arreglos

Formas de Implementar arreglos en Python

Python es un lenguaje que propone varias formas de abordar el tema

❑ Mediante la utilización de estructuras de datos

- Las estructuras de datos en Python se pueden entender como un tipo de dato compuesto, debido a que en una misma variable podemos almacenar una estructura completa con información. Dichas estructuras, pueden tener diferentes características y funcionalidades. De hecho, existen múltiples tipos de estructuras de datos en Python. Por ejemplo Listas, Cadenas, Diccionarios, Conjuntos, Tuplas donde las características más notables pueden ser:
 - heterogéneas: pueden estar conformadas por elementos de distintos tipos, incluidos otras listas.
 - mutables: sus elementos pueden modificarse.

❑ Mediante la importación de la librería NumPy

- Es una biblioteca para el lenguaje de programación Python que da soporte para crear vectores y matrices grandes multidimensionales, junto con una gran colección de funciones matemáticas de alto nivel para operar con ellas. (Wikipedia)

❑ Mediante la utilización de la librería <arrays>

- El módulo array de la librería estándar de Python permite declarar un objeto que es similar a una lista pero sólo puede almacenar datos del mismo tipo: números enteros con distintos tamaños y números de punto flotante, entre otros. Los arreglos son tipos de secuencia y se comportan de manera similar a las listas, excepto que el tipo de objetos almacenados en ellos está restringido.

Vectores 1D (arrays Library)

Qué podemos hacer con los Arreglos en Python mediante el uso de la librería <arrays>

Las operaciones o actividades que podemos realizar con los arreglos son muchas. Algunas de las más representativas pueden ser las siguientes:

- Crear el arreglo
- Inicializarlo
- Insertar elementos
- Agregar elementos al final
- Eliminación de elementos
- Recorrido del arreglo
- Escritura del arreglo
- Eliminación del arreglo
- Iterar un arreglo
- Búsqueda de un elemento
- Etc....

Vectores 1D (arrays Library)

Arreglos 1D (Unidimensionales)

Un arreglo **unidimensional** es una colección, o grupo de datos, donde:

Cada dato tiene su posición (primero, segundo, tercero...), todos los datos del grupo son del mismo tipo, es decir, o todos son enteros, o todos son reales, etc. y coexisten en una **única fila**. El anterior ejemplo nos muestra un vector unidimensional de 3 posiciones para almacenar información y la variable se llama datos. Cada caja representa un dato del arreglo o un **elemento**.

Dimension miVector(3)

miVector	2	3	4
laPosicion	1	2	3

Vectores 1D

**Implementados con
<arrays library>
en Python**

Vectores 1D (arrays Library)

Acceso a sus Datos

¿Cómo accedemos a uno de esos datos?

Usamos el nombre del arreglo y el índice que identifica al elemento: nombre_arreglo[índice]

Asignación de Datos

Si se desea asignar el valor de 2 al primer elemento del arreglo y 9 al tercero:

- `miVector[1] = 2`
- `miVector[3] = 9`

Algoritmo EjemploVector

Dimension `miVector(3)`

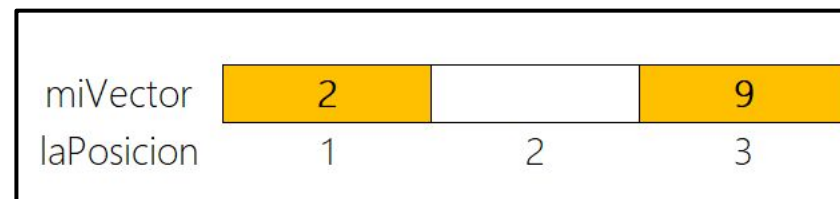
`miVector(1)=2`

`miVector(3)=9`

Escribir "Ver miVector en: ",`miVector(1)`

Escribir "Ver miVector en: ",`miVector(3)`

FinAlgoritmo



Dirección Memoria	Valor	Representación	Tamaño
100	2	miVector(1)	4
104	Desconocido	miVector(2)	4
108	9	miVector(3)	4

Vectores 1D - Creación

Creación de Arreglos en Python mediante el uso de la librería <arrays>

Aquí, creamos una matriz del tipo <float>.
La letra “d” es un código tipo. Esto determina el tipo de arreglo durante la creación.

```
import array as arr
a = arr.array('d', [1.1, 3.5, 4.5])
print(a)
```

Commonly used type codes are listed as follows:

Code	C Type	Python Type	Min bytes
b	signed char	int	1
B	unsigned char	int	1
u	Py_UNICODE	Unicode	2
h	signed short	int	2
H	unsigned short	int	2
i	signed int	int	2
I	unsigned int	int	2
l	signed long	int	4
L	unsigned long	int	4
f	float	float	4
d	double	float	8

Vectores 1D - Acceso

Accesos de Arreglos en Python mediante el uso de la librería <arrays>

Aquí, creamos un arreglo del tipo <int>.
La letra "i" es un código tipo. Esto determina el tipo de arreglo durante la creación.

```
import array as arr
a = arr.array('i', [2, 4, 6, 8])

print("First element:", a[0])
print("Second element:", a[1])
print("Last element:", a[-1])
```

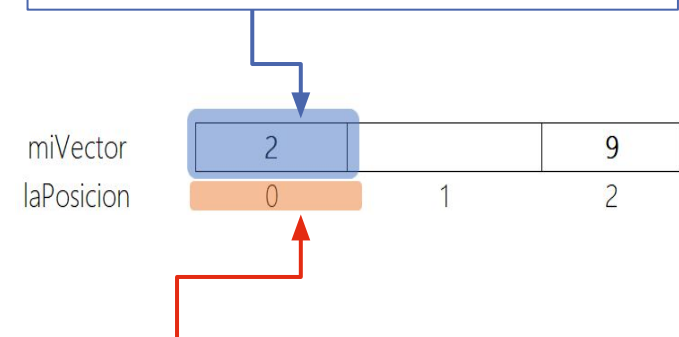
Output

```
First element: 2
Second element: 4
Last element: 8
```

Note: The index starts from 0 (not 1) similar to lists.

Referencia al valor contenido

En este ejemplo indicado por la <miVector [laPosicion]>



Referencia a la posición o índice

En este ejemplo indicado por la variable <laPosicion>

Vectores 1D - Slicing

Slicing de Arreglos en Python mediante el uso de la librería <arrays>

“Slicing” Es la posibilidad de acceder de manera directa a diferentes posiciones consecutivas del arreglo. Aquí se presentan algunos ejemplos de “Slicing” y en contrapartida cómo debería haberse escrito la funcionalidad en caso de no haber tenido esta funcionalidad particular de este lenguaje.

```
import array as arr

numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
numbers_array = arr.array('i', numbers_list)

print(numbers_array[2:5]) # 3rd to 5th
print(numbers_array[:5]) # beginning to 4th
print(numbers_array[5:]) # 6th to end
print(numbers_array[:]) # beginning to end
```

```
1 import array as arr
2
3 numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
4 numbers_array = arr.array('i', numbers_list)
5 for i in range(0,8,1):
6     print(numbers_array[i])
```

```
1 import array as arr
2
3 numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
4 numbers_array = arr.array('i', numbers_list)
5 for i in range(2,5,1):
6     print(numbers_array[i])
```

```
1 import array as arr
2
3 numbers_list = [2, 5, 62, 5, 42, 52, 48, 5]
4 numbers_array = arr.array('i', numbers_list)
5 for i in range(0,3,1):
6     print(numbers_array[i])
```

Vectores 1D - Reemplazo

Cambiando e Insertando Elementos en arreglos en Python mediante el uso de la librería <arrays>

El reemplazo de los valores en un arreglo se debe hacer mediante la referencia del nombre del vector y el elemento a reemplazar.

```
1 Algoritmo ChangingVector
2   Dimension mivector(6)
3   mivector[1] = 10
4   mivector[2] = 20
5   mivector[3] = 30
6   mivector[4] = 0
7   mivector[5] = 0
8   mivector[6] = 0
9   Escribir "Valor1: ",mivector[1]
10  Escribir "Valor2: ",mivector[2]
11  Escribir "Valor3: ",mivector[3]
12  Escribir "Valor4: ",mivector[4]
13  Escribir "Valor5: ",mivector[5]
14  Escribir "Valor6: ",mivector[6]
15  //REEMPLAZO
16  mivector[1] = 100
17  mivector[3] = 300
18  Escribir "Reemplazo Valor1: ",mivector[1]
19  Escribir "Reemplazo Valor3: ",mivector[3]
20 FinAlgoritmo
```

```
import array as arr

numbers = arr.array('i', [1, 2, 3, 5, 7, 10])

# changing first element
numbers[0] = 0
print(numbers)    # Output: array('i', [0, 2, 3, 5, 7, 10])

# changing 3rd to 5th element
numbers[2:5] = arr.array('i', [4, 6, 8])
print(numbers)    # Output: array('i', [0, 2, 4, 6, 8, 10])
```

changing 3rd to 5th element with iterations

```
17 # changing 3rd to 5th element with iterations
18 for i in range(2,5,1):
19     value=int(input("Ingrese Valor a reemplazar en (" +str(i)+")"))
20     numbers[i]=value
21
22 print(numbers)
23
24
```

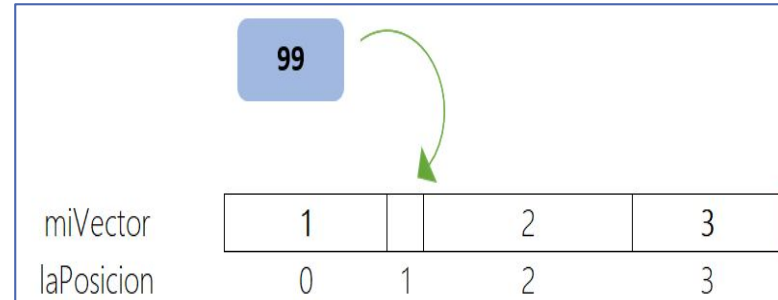
Vectores 1D - Inserción

Inserción Elementos en arreglos en Python mediante el uso de la librería <arrays>

Insert permite indicar de manera específica la posición y elemento a insertar. Una vez realizada la inserción se realiza el desplazamiento a derecha de los elementos de izquierda (en caso existan)

main.py

```
1 import array as arr
2
3 numbers = arr.array('i', [1, 2, 3])
4
5 numbers.insert(1,99)
6 print(numbers)      # Output: array('i', [1, 99, 2, 3])
7
8 numbers.append(4)
9 print(numbers)      # Output: array('i', [1, 99, 2, 3, 4])
10
11
```



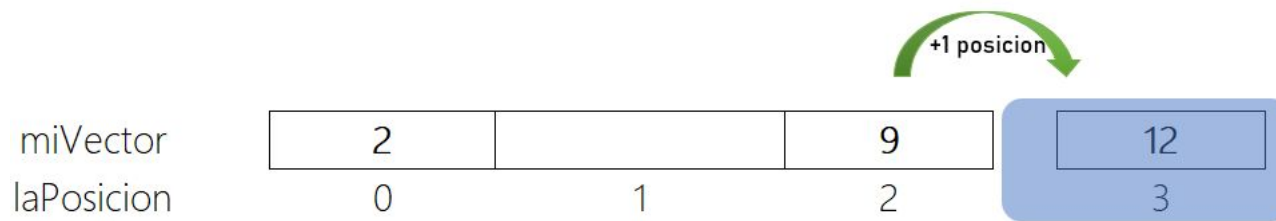
miVector	1	99	2	3	4
laPosicion	0	1	2	3	4

El <Insert> del valor 99 desplaza hacia derecha a los elementos existentes. Adicionalmente se incorpora por <append> un 4 al final del vector, para verificar la diferencia entre ambos.

Vectores 1D – Adición o Append

Adicionando Elementos en arreglos en Python mediante el uso de la librería <arrays>

Append permite la adición de un elemento al final del arreglo. Extend tiene el mismo propósito pero para mas de un elemento.



```
import array as arr

numbers = arr.array('i', [1, 2, 3])

numbers.append(4)
print(numbers)    # Output: array('i', [1, 2, 3, 4])

# extend() appends iterable to the end of the array
numbers.extend([5, 6, 7])
print(numbers)    # Output: array('i', [1, 2, 3, 4, 5, 6, 7])
```

Vectores 1D - Iterando

Lectura /Escritura de Valores desde una estructura iterativa

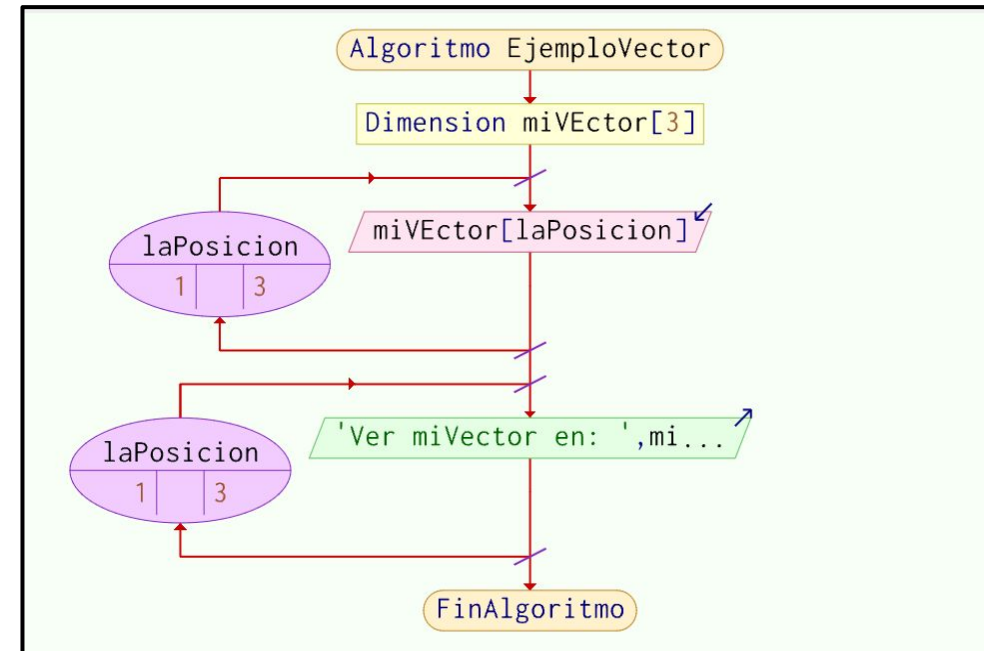
```

Algoritmo EjemploVector
  Dimension miVector(3)

  Para laPosicion=1 hasta 3
  |   Leer miVector(laPosicion)
  FinPara

  Para laPosicion=1 hasta 3
  |   Escribir "Ver miVector en: ",miVector(laPosicion)
  FinPara

FinAlgoritmo
  
```



Vectores 1D - Iterando

Lectura /Escritura de Valores desde una estructura iterativa (for in range)

```
import array as arr

# array with int type
miVectorA = arr.array('i', [0, 0, 0])

print ("miVectorA - Lectura desde for in range: ", end = "\n")
for i in range (0, 3):
    valor=int(input ("Ingrese miVectorA("+str(i)+"): "))
    miVectorA[i]=valor
print()

print ("miVectorA - Escritura desde for in range: ", end = " ")
for i in range (0, 3):
    print (miVectorA[i], end = " ")
print()
```


Vectores 1D - Iterando

Lectura /Escritura de Valores desde una estructura iterativa (for in <arreglo, lista, colección))

```
import array as arr

# array with int type

miVectorB = arr.array('i', [0, 0, 0])

print ("miVectorB -Lectura desde for in arreglo: ", end = "\n")
j=0
for i in (miVectorB):
    valor=int(input ("Ingrese miVectorB("+str(j)+"): "))
    miVectorB[j]=valor
    j=j+1
print()

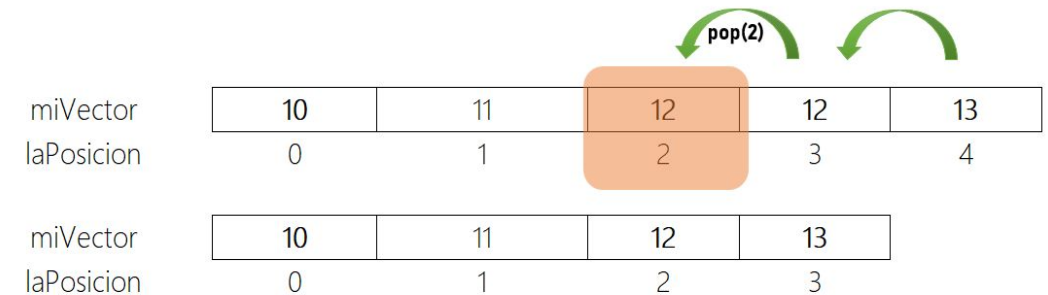
print ("miVectorB -Escritura desde for in arreglo: ", end = " ")
for i in (miVectorB):
    print (i, end = " ")
print()
```

Vectores (Eliminación)

Eliminación de Elementos en arreglos en Python mediante el uso de la librería <arrays>

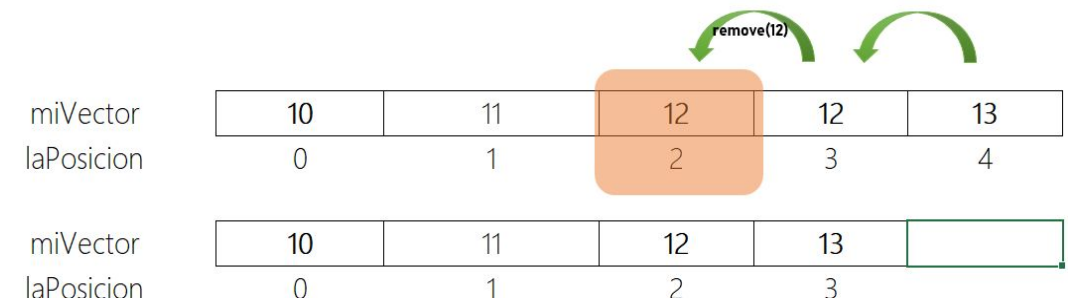
Eliminación mediante el índice (posición del elemento) del y pop

```
main.py
1 import array as arr
2
3 numbers = arr.array('i', [10, 11, 12, 12, 13])
4
5 print(numbers.pop(2)) # Output: 12
6 del numbers[0] # Remove: 10
7 print(numbers) # Output: array('i', [11, 12, 13])
```



Eliminación mediante el contenido de la posición (remove)

```
main.py
1 import array as arr
2
3 numbers = arr.array('i', [10, 11, 12, 12, 13])
4
5 numbers.remove(12)
6 print(numbers) # Output: array('i', [10, 11, 12, 13])
7
8
```



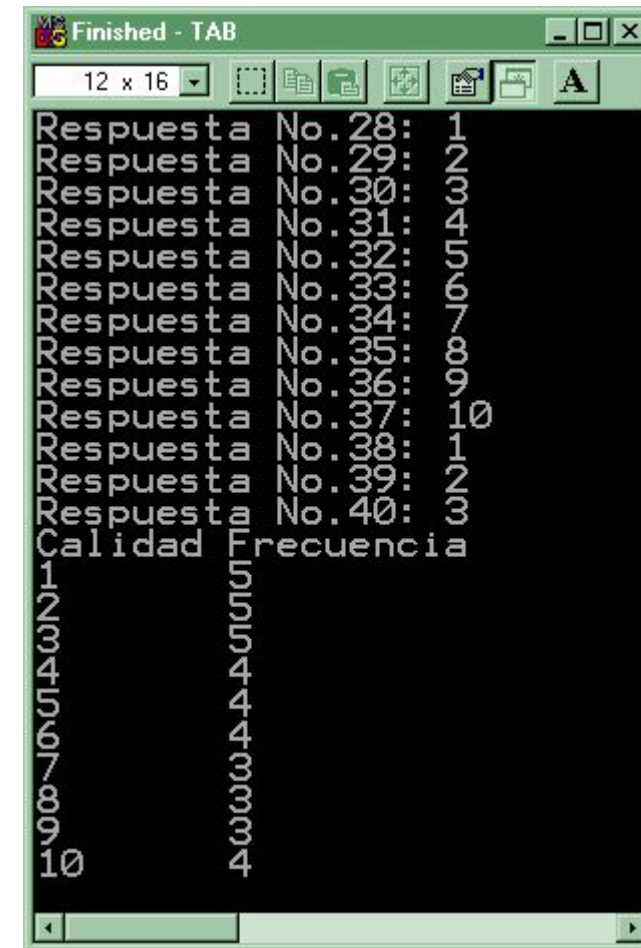
Ejercicio

Trabajo para Casa

En lugar de almacenar valores en los elementos de un arreglo, algunas aplicaciones usan los arreglos para almacenar “índices”, que permiten llevar estadísticas, de un dato en particular.

Por ejemplo:

Se pidió a 40 estudiantes que calificaran la calidad de la comida de la cafetería en una escala del 1 al 10 (1 es terrible y 10 excelente).



Calidad	Frecuencia
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10

Vectores 2D o Matrices

**Matrices 2D
Implementadas con
<Numpy library>
en Python**

Vectores 2D o Matrices

Cuando hablamos de matrices en el mundo de la programación te puedes imaginar una tabla con filas y columnas que contiene los mismos tipos de datos. En sí es como tener una reunión de vectores o arreglos, uno debajo del otro. En un array unidimensional o vector cada elemento se referencia por un índice, en un array multidimensional, cada elemento se va a referenciar por 2 o más índices, y ahora la representación lógica ya no va a ser un vector, sino una matriz

ARREGLOS MULTIDIMENSIONALES					
		columnas			
		0	1	2	3
filas	0	celda (0,0)	celda (0,1)	celda (0,2)	celda (0,3)
	1	celda (1,0)	celda (1,1)	celda (1,2)	celda (1,3)
	2	celda (2,0)	celda (2,1)	celda (2,2)	celda (2,3)
	3	celda (3,0)	celda (3,1)	celda (3,2)	celda (3,3)

Vectores 2D o Matrices

Acceso a sus Datos

¿Cómo accedemos a uno de esos datos?

Usamos el nombre del arreglo y el índice que identificando fila y columna: `nombre_arreglo[indiceFila, indiceColumna]`

Asignación de Datos

- `miMatriz[0,2] = 2`
- `miMatriz[1,1] = 9`

ARREGLOS MULTIDIMENSIONALES				
columnas				
filas	0	1	2	3
	0 celda (0,0)	celda (0,1)	2	celda (0,3)
	1 celda (1,0)	9	celda (1,2)	celda (1,3)
	2 celda (2,0)	celda (2,1)	celda (2,2)	celda (2,3)
	3 celda (3,0)	celda (3,1)	celda (3,2)	celda (3,3)

Vectores 2D o Matrices

Dimensionamiento, Lectura y Escritura

```
1 Algoritmo Arreglo2DMatriz
2   Dimension Mat[4,4];
3   FIL = 4
4   COL = 4
5   Para i=1 Hasta FIL Con Paso 1 Hacer
6       Para j=1 Hasta COL Con Paso 1 Hacer
7           Escribir "Ingrese un NUmero"
8           Leer Mat[i,j]
9       FinPara
10  Fin Para
11
12
13  Para i=1 Hasta FIL Con Paso 1 Hacer
14      Escribir ""
15      Para j=1 Hasta COL Con Paso 1 Hacer
16          Escribir " "+Mat[i,j] Sin Saltar
17      FinPara
18  Fin Para
19  Escribir ""
20 FinAlgoritmo
21
```

Dimensionamiento de cantidad de filas y columnas

El for i establece la fila y j itera 4 veces por cada i.
La primera **lectura** de los valores referencia a (1,1) (1,2) (1,3) (1,4) Es decir i=1 y J, 1,2,3,4.
Luego i = 2 y nuevamente j 1,2,3,4. Asi hasta completar la matriz

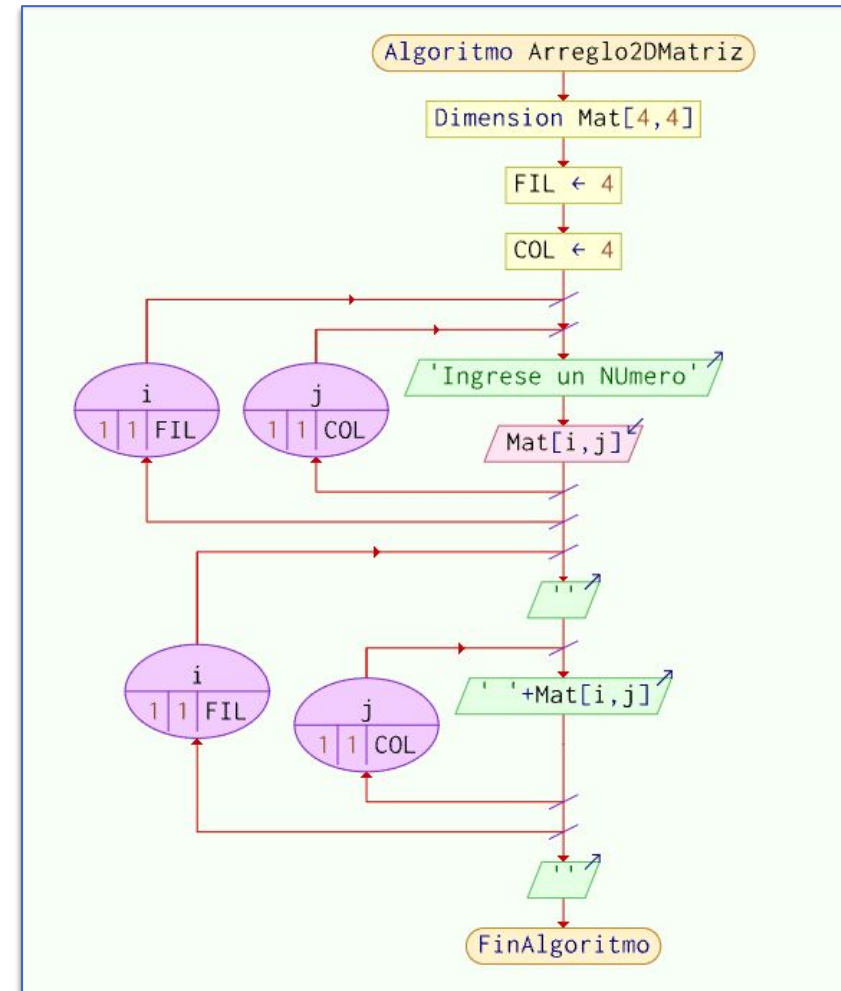
El for i establece la fila y j itera 4 veces por cada i.
La primera **Escritura** de los valores referencia a (1,1) (1,2) (1,3) (1,4) Es decir i=1 y J, 1,2,3,4.
Luego i = 2 y nuevamente j 1,2,3,4. Asi hasta completar la matriz

Matrices o Vectores 2D

```

1  Algoritmo Arreglo2DMatriz
2    Dimension Mat[4,4];
3    FIL = 4
4    COL = 4
5    Para i=1 Hasta FIL Con Paso 1 Hacer
6      Para j=1 Hasta COL Con Paso 1 Hacer
7        Escribir "Ingrese un NUmero"
8        Leer Mat[i,j]
9      FinPara
10   Fin Para

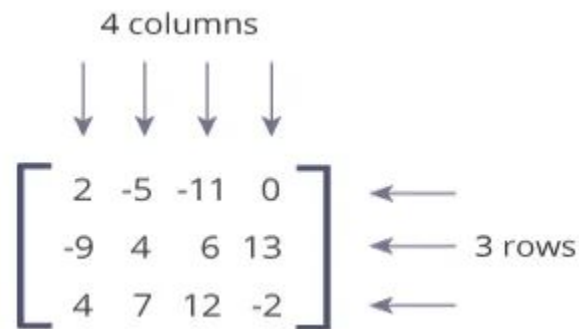
11
12
13   Para i=1 Hasta FIL Con Paso 1 Hacer
14     Escribir ""
15     Para j=1 Hasta COL Con Paso 1 Hacer
16       Escribir " "+Mat[i,j] Sin Saltar
17     FinPara
18   Fin Para
19   Escribir ""
20 FinAlgoritmo
21
  
```



Vectores 2D o Matrices

Características para implementarlas en Python

Python no tiene un tipo incorporado (built-in) para matrices. Las matrices deben derivarse del uso de librerías externas o de la utilización de listas de listas. De estas dos alternativas la más cercana al concepto puro de arreglos de dos dimensiones es la que presenta a la librería **NUMPY** como alternativa.



Vectores 2D o Matrices

Implementación mediante Lista de Listas

Aquí la siguiente declaración implica la definición de una Lista de 2 filas y tres columnas.

```
A = [[1, 4, 5],  
     [-5, 8, 9]]
```

$$\begin{bmatrix} 1 & 4 & 5 \\ -5 & 8 & 9 \end{bmatrix}$$

Vectores 2D o Matrices

Implementación mediante Numpy

`Numpy.array` vs `Numpy.matrix`

La librería Numpy permite la creación de matrices mediante la utilización de dos métodos. Uno es `Numpy.array` y otro `Numpy.matrix`.

Las `numpy.matrix` son estrictamente bidimensionales, mientras que las `numpy.array` son N-dimensionales. Los objetos `matrix` son una subclase de `ndarray`, por lo que heredan todos los atributos y métodos de `ndarrays`.

La principal ventaja de las `numpy.matrix` es que proporcionan una notación conveniente para la multiplicación de matrices: si `a` y `b` son matrices, entonces `a*b` es su producto matricial.

La principal ventaja de utilizar `numpy arrays` es que son más genéricos que las estrictamente 2 dimensiones que permite la `matrix`.

Vectores 2D o Matrices

Inicialización utilizando Numpy.array

El primer caso define una matriz de 2 filas y tres columnas. No especifica el tipo pero infiere datos enteros.

En el segundo caso es igual al primero pero siendo explícito en qué tipo de datos quiero que guarde la matriz.

En el tercer caso define una matriz de 2 filas y tres columnas. Al colocar en A[0][0] el valor 1.1 y sabiendo que los arreglos manejan solo un tipo de dato, toda la matriz se define como float. No especifica el tipo pero infiere datos float.

```
import numpy as np
A = np.array([[1, 2, 3], [3, 4, 5]])
print(A)
A = np.array([[1, 2, 3], [3, 4, 5]], dtype = int)
print(A)
A = np.array([[1.1, 2, 3], [3, 4, 5]]) # Array of floats
print(A)
```

Vectores 2D o Matrices

Inicialización utilizando Numpy.array

El primer caso "matrix1" define 2x3 con valores explícitos en 0. Asume tipo int

El segundo caso "matrix2" utiliza el método zeros completando en 2x3 con valores. Infiere 0 float y por ello guarda 0.

En "matrix3" inicializa todo con 1 entero porque fuimos explícitos en el dtype

En "matrix4" inicializa 2x3 con valores 0,1,2 ya que usa range.

```
import numpy as np

matrix1 = np.array([[0,0,0],[0,0,0]])
print(matrix1)

matrix2 = np.zeros([2,3])
print(matrix2)

matrix3 = np.ones([2,3],dtype=int)
print(matrix3)

matrix4 = np.array([range(3),range(3)])
print("range:",matrix4)
```

Vectores 2D o Matrices

Otras formas de Inicialización utilizando Numpy.matrix

Tres formas diferentes de dimensionar e inicializar una matriz de 3x3.

#Formas Inicializar 1

```
matrix1 = np.matrix([range(3), range(3),range(3)])  
print('Forma 1',matrix1)
```

#Formas Inicializar 2

```
matrix1 = np.matrix('1 2 3; 4 5 6; 7 8 9')  
print ('Forma 2',matrix1)
```

#Formas Inicializar 3

```
matrix1 = np.matrix([[1,2,3],[4,5,6],[7,8,9]])  
print ('Forma 3',matrix1)
```

Vectores 2D o Matrices

Acceso de los elementos de la Matriz

El acceso a los elementos de la matriz requiere de forma mandatoria la referencia de las filas y columnas a las cuales se quiere acceder

```
import numpy as np

A = np.array([[1, 4, 5, 12],
              [-5, 8, 9, 0],
              [-6, 7, 11, 19]])

# First element of first row
print("A[0][0] =", A[0][0])

# Third element of second row
print("A[1][2] =", A[1][2])

# Last element of last row
print("A[-1][-1] =", A[-1][-1])
```

Vectores 2D o Matrices

Recorrido de los elementos de la Matriz

El recorrido tradicional implica la utilización de dos estructuras iterativas. Una para determinar la fila y otra para determinar la columna.

```
import numpy as np
matrix = np.array([[1.1, 2, 3], [3, 4, 5]])

filas=len(matrix)
columnas=len(matrix[0])
for i in range(0,filas,1):
    for j in range(0, columnas, 1):
        print("Elemento:",matrix[i][j])
```

```
Elemento: 1.1
Elemento: 2.0
Elemento: 3.0
Elemento: 3.0
Elemento: 4.0
Elemento: 5.0
> |
```


Conclusiones

PYTHON LISTS VS NUMPY ARRAYS

NumPy es el paquete fundamental para la computación científica en Python. Las matrices con NumPy facilitan operaciones matemáticas avanzadas y de otro tipo en grandes cantidades de datos. Por lo general, estas operaciones se ejecutan de manera más eficiente y con menos código de lo que es posible usando las secuencias integradas de Python. NumPy no es otro lenguaje de programación sino un módulo de extensión de Python. Proporciona operaciones rápidas y eficientes en matrices de datos homogéneos.

Algunos puntos importantes sobre Numpy:

Podemos crear una matriz N-dimensional en Python usando `numpy.array()` o `numpy.matrix()`

- Las matrices son homogéneas por defecto, lo que significa que los datos dentro de una matriz deben ser del mismo tipo de datos. (Tenga en cuenta que también puede crear una matriz estructurada en Python).
- Es posible un funcionamiento inteligente.
- Numpy array tiene diversas funciones, métodos y variables para facilitar nuestra tarea de cálculo de matrices.
- Los elementos de una matriz se almacenan de forma continua en la memoria. Por ejemplo, todas las filas de una matriz de dos dimensiones deben tener el mismo número de columnas. O una matriz de tres dimensiones debe tener el mismo número de filas y columnas.

Conclusiones

¿Ventajas o no tanto?

Las listas son mucho más flexibles que los arrays. Pueden almacenar elementos de diferentes tipos de datos, incluidas cadenas. Y, si necesita hacer cálculos matemáticos en arreglos y matrices, es mucho mejor que use algo como NumPy .

Entonces, ¿cuáles son los usos de los arreglos creados a partir del módulo de `<array>` de Python?

El `<array.array>` tipo es solo un “envoltorio” sobre arreglos en C que proporciona un almacenamiento eficiente en el espacio (tipos de datos básicos de estilo C). Si necesita asignar una matriz que sabe que no cambiará, entonces las arrays pueden ser más rápidas y usar menos memoria que las listas.

Vectores 2D o Matrices

A jugar con las Matrices

Recorrido por las diferentes partes que nos ofrece una Matriz. Diagonal Principal, Contra Diagonal, Partes superiores e inferiores (izquierdas o derechas)

ARREGLOS MULTIDIMENSIONALES

columnas

	0	1	2	3
0	celda (0,0)	celda (0,1)	celda (0,2)	celda (0,3)
1	celda (1,0)	celda (1,1)	celda (1,2)	celda (1,3)
2	celda (2,0)	celda (2,1)	celda (2,2)	celda (2,3)
3	celda (3,0)	celda (3,1)	celda (3,2)	celda (3,3)

Para recorrer un arreglo multidimensional necesito (2) dos for.

Para trabajar con la diagonales el arreglo multidimensional debe tener el mismo numero de fila y columna

Diagonal Principal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Si observan: La diag. Ppal se obtiene cuando la fila coincide con la columna

fila	columna
0	0
1	1
2	2
3	3

Contradiagonal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Si observan: La contradiagonal se obtiene cuando la suma de la fila + la columna da el tamaño

fila	columna
0	3
1	2
2	1
3	0

Como se puede recorrer usando 2 (dos) for

Parte superior de la Diag. Ppal
Sin considerar la diag. Ppal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Observamos la parte superior e inferior de la diag. Ppal

Parte superior de la Diag. Ppal considerando la diag. Ppal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte inferior de la Diag. Ppal
Sin considerar la diag. Ppal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte inferior de la Diag. Ppal considerando la diag. Ppal

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte Superior de la Contradiag
Sin considerar la contradiag.

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte Superior de la Contradiag considerando la contradiag.

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte Inferior de la Contradiag
Sin considerar la contradiag.

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Parte Inferior de la Contradiag considerando la contradiag.

	0	1	2	3
0	(0,0)	(0,1)	(0,2)	(0,3)
1	(1,0)	(1,1)	(1,2)	(1,3)
2	(2,0)	(2,1)	(2,2)	(2,3)
3	(3,0)	(3,1)	(3,2)	(3,3)

Observamos la parte superior e inferior de la diag. Ppal

Gracias

