Tema 0 - Programación básica con Python IV

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

Contenido

Contenido

Contenido

Contenido

Contenido

Productos entre arreglos

Recordemos que vector es sinónimo de arreglo de una dimensión, y matriz es sinónimo de arreglo de dos dimensiones.

Producto interno (vector-vector)

El producto interno entre dos vectores se obtiene usando la función `dot` provista por NumPy:

```
>>> a = array([-2.8 , -0.88,  2.76,  1.3 ,  4.43])
>>> b = array([ 0.25, -1.58,  1.32, -0.34, -4.22])
```

Producto interno (vector-vector)

El producto interno entre dos vectores se obtiene usando la función `dot` provista por NumPy:

```
>>> a = array([-2.8 , -0.88,  2.76,  1.3 ,  4.43])
>>> b = array([ 0.25, -1.58,  1.32, -0.34, -4.22])
>>> dot(a, b)
```

Producto interno (vector-vector)

El producto interno entre dos vectores se obtiene usando la función `dot` provista por NumPy:

```
>>> a = array([-2.8 , -0.88,  2.76,  1.3 ,  4.43])
>>> b = array([ 0.25, -1.58,  1.32, -0.34, -4.22])
>>> dot(a, b)
-14.803
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar 1400
```

También se usa para calcular promedios ponderados:

```
>>> notas = array([45, 98, 32])
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar 1400
```

También se usa para calcular promedios ponderados:

```
>>> notas = array([45, 98, 32])
>>> ponderaciones = array([30, 30, 40]) / 100.
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar 1400
```

También se usa para calcular promedios ponderados:

```
>>> notas = array([45, 98, 32])
>>> ponderaciones = array([30, 30, 40]) / 100.
>>> nota_final = dot(notas, ponderaciones)
```

El producto interno es una operación muy común. Por ejemplo, suele usarse para calcular totales:

```
>>> precios = array([200, 100, 500, 400, 400, 150])
>>> cantidades = array([1, 0, 0, 2, 1, 0])
>>> total_a_pagar = dot(precios, cantidades)
>>> total_a_pagar 1400
```

También se usa para calcular promedios ponderados:

```
>>> notas = array([45, 98, 32])
>>> ponderaciones = array([30, 30, 40]) / 100.
>>> nota_final = dot(notas, ponderaciones)
>>> nota_final
55.7
```

Producto matriz-vector

El producto matriz-vector es el vector de los productos internos. El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función `dot` entre las filas de la matriz y el vector:

```
>>> a = array([[-0.6,  4.8, -1.2],
               [-2. , -3.6, -2.1],
               [ 1.7,  4.9,  0. ]])
>>> x = array([-0.6, -2. ,  1.7])
```

Producto matriz-vector

El producto matriz-vector es el vector de los productos internos. El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función `dot` entre las filas de la matriz y el vector:

```
>>> a = array([[-0.6,  4.8, -1.2],
               [-2. , -3.6, -2.1],
               [ 1.7,  4.9,  0. ]])
>>> x = array([-0.6, -2. ,  1.7])
>>> dot(a, x)
```

Producto matriz-vector

El producto matriz-vector es el vector de los productos internos. El producto matriz-vector puede ser visto simplemente como varios productos internos calculados de una sola vez.

Esta operación también es obtenida usando la función `dot` entre las filas de la matriz y el vector:

```
>>> a = array([[-0.6,  4.8, -1.2],
               [-2. , -3.6, -2.1],
               [ 1.7,  4.9,  0. ]])
>>> x = array([-0.6, -2. ,  1.7])
>>> dot(a, x)
array([-11.28,   4.83, -10.82])
```

Producto matriz-matriz

El producto matriz-matriz es la matriz de los productos internos entre las filas de la primera matriz y las columnas de la segunda.

```
>>> a = array([[ 2,  8],
               [-3,  7],
               [-8, -5]])
>>> b array([[-3, -5, -6, -3],
             [-9, -2,  3, -3]])
```

Producto matriz-matriz

El producto matriz-matriz es la matriz de los productos internos entre las filas de la primera matriz y las columnas de la segunda.

```
>>> a = array([[ 2,  8],
               [-3,  7],
               [-8, -5]])
>>> b array([[-3, -5, -6, -3],
             [-9, -2,  3, -3]])
>>> dot(a, b)
```

Producto matriz-matriz

El producto matriz-matriz es la matriz de los productos internos entre las filas de la primera matriz y las columnas de la segunda.

```
>>> a = array([[ 2,  8],
               [-3,  7],
               [-8, -5]])
>>> b array([[-3, -5, -6, -3],
             [-9, -2,  3, -3]])
>>> dot(a, b)
array([[-78, -26,  12, -30],
       [-54,   1,  39, -12],
       [ 69,  50,  33,  39]])
```

La multiplicación de matrices puede ser vista como varios productos matriz-vector (usando como vectores todas las filas de la segunda matriz), calculados de una sola vez.

En resumen, al usar la función `dot`, la estructura del resultado depende de cuáles son los parámetros pasados:

1. `dot`(vector, vector) $\rightarrow$ número.

En resumen, al usar la función `dot`, la estructura del resultado depende de cuáles son los parámetros pasados:

1. `dot`(vector, vector) $\rightarrow$ número.
2. `dot`(matriz, vector) $\rightarrow$ vector.

En resumen, al usar la función `dot`, la estructura del resultado depende de cuáles son los parámetros pasados:

1. `dot`(vector, vector) $\rightarrow$ número.
2. `dot`(matriz, vector) $\rightarrow$ vector.
3. `dot`(matriz, matriz) $\rightarrow$ matriz.

Contenido

Solución de sistemas lineales

Un problema recurrente en Ciencias consiste en obtener cuál es el vector x cuando A y b son dados:

$$Ax = b$$

La ecuación matricial $Ax = b$ es una manera abreviada de expresar un sistema de ecuaciones lineales. Por ejemplo, la ecuación del diagrama es equivalente al siguiente sistema de tres ecuaciones que tiene las tres incógnitas w, y y z:

$$\begin{aligned} 36w + 51y + 13z &= 3 \\ 52w + 34y + 74z &= 45 \\ 7y + 1.1z &= 33 \end{aligned}$$

Este sistema se representa matricialmente:

$$\begin{bmatrix} 36 & 51 & 13 \\ 52 & 34 & 74 \\ & 7 & 1.1 \end{bmatrix} \begin{bmatrix} w \\ y \\ z \end{bmatrix} = \begin{bmatrix} 3 \\ 45 \\ 33 \end{bmatrix}$$

La teoría detrás de la solución de problemas de este tipo, se puede consultar en cualquier texto de álgebra lineal. Sin embargo, como este tipo de problemas aparece a menudo en la práctica, aprenderemos cómo obtener rápidamente la solución usando Python.

Dentro de los varios módulos incluidos en NumPy, está el módulo `numpy.linalg`, que provee algunas funciones que implementan algoritmos de álgebra lineal. Dentro de este módulo está la función `solve`, que entrega la solución x de un sistema a partir de la matriz A y el vector b:

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
>>> x
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
>>> x
array([-7.10829222,  4.13213834,  3.70457422])
```

Podemos ver que el vector x en efecto satisface la ecuación $Ax = b$:

```
>>> dot(a, x)
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
>>> x
array([-7.10829222,  4.13213834,  3.70457422])
```

Podemos ver que el vector x en efecto satisface la ecuación $Ax = b$:

```
>>> dot(a, x)
array([  3.,  45.,  33.])
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
>>> x
array([-7.10829222,  4.13213834,  3.70457422])
```

Podemos ver que el vector x en efecto satisface la ecuación $Ax = b$:

```
>>> dot(a, x)
array([  3.,  45.,  33.])
>>> b
```

```
>>> a = array([[ 36. ,  51. ,  13. ],
...            [ 52. ,  34. ,  74. ],
...            [  0. ,   7. ,   1.1]])
>>> b = array([  3.,  45.,  33.])
>>> x = solve(a, b)
>>> x
array([-7.10829222,  4.13213834,  3.70457422])
```

Podemos ver que el vector x en efecto satisface la ecuación $Ax = b$:

```
>>> dot(a, x)
array([  3.,  45.,  33.])
>>> b
array([  3.,  45.,  33.])
```

Sin embargo, es importante tener en cuenta que los valores de tipo real casi nunca están representados de manera exacta en la solución numérica, y que el resultado de un algoritmo que involucra muchas operaciones puede sufrir de algunos errores de redondeo. Por esto mismo, puede ocurrir que aunque los resultados se vean iguales en la consola, los datos obtenidos son sólo aproximaciones y no exactamente los mismos valores:

```
>>> (dot(a, x) == b).all()
```

Sin embargo, es importante tener en cuenta que los valores de tipo real casi nunca están representados de manera exacta en la solución numérica, y que el resultado de un algoritmo que involucra muchas operaciones puede sufrir de algunos errores de redondeo. Por esto mismo, puede ocurrir que aunque los resultados se vean iguales en la consola, los datos obtenidos son sólo aproximaciones y no exactamente los mismos valores:

```
>>> (dot(a, x) == b).all()
False
```

Contenido

Otras funciones dentro de `numpy.linalg`

Para extender (y simplificar el trabajo para codificar) el manejo de arreglos en Python, se cuenta con otras funciones que se ocupan de manera continua.

Como se ha mencionado anteriormente, es necesario repasar el álgebra lineal básica para tener en cuenta el proceso con el cual, Pyhon devuelve una respuesta.

Función Eye

La función `eye` genera una matriz $N \times N$ diagonal con `eye(N)`, pero admite otros parámetros que permiten hacer matrices no cuadradas, tener otro tipo de datos y hacer distinto de 0 otra diagonal diferente.

```
>>>eye(2)
```

Función Eye

La función `eye` genera una matriz $N \times N$ diagonal con `eye(N)`, pero admite otros parámetros que permiten hacer matrices no cuadradas, tener otro tipo de datos y hacer distinto de 0 otra diagonal diferente.

```
>>>eye(2)
array([[ 1.,  0.],
       [ 0.,  1.]])
```

```
>>> eye(2,3)
```

```
>>> eye(2,3)
```

```
>>> eye(2,3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

```
>>> eye(2,3,k=1)
```

```
>>> eye(2,3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

```
>>> eye(2,3,k=1)
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

```
>>> eye(2,3,k=1,dtype=complex)
```

```
>>> eye(2,3)
array([[ 1.,  0.,  0.],
       [ 0.,  1.,  0.]])
```

```
>>> eye(2,3,k=1)
array([[ 0.,  1.,  0.],
       [ 0.,  0.,  1.]])
```

```
>>> eye(2,3,k=1,dtype=complex)
array([[ 0.+0.j,  1.+0.j,  0.+0.j],
       [ 0.+0.j,  0.+0.j,  1.+0.j]])
```

Función `reshape`

La función `reshape` permite cambiar las dimensiones de una matriz, siempre respetando el número total de elementos.

No cambia el objeto original, pero devuelve otro objeto que apunta los mismos datos, de forma que si modificamos uno, el otro lo hará también.

```
>>> a = random.rand(4,4)
```

```
>>> a = random.rand(4,4)
>>> a
array([[ 0.51878337,  0.93337481,  0.84368137,  0.07324918],
       [ 0.12929511,  0.92344357,  0.50366378,  0.59754141],
       [ 0.67841199,  0.73959186,  0.45789404,  0.85003645],
       [ 0.95552903,  0.81794353,  0.78810869,  0.05192744]])
```

```
>>> b = reshape(a,(2,8))
```

```
>>> b = reshape(a,(2,8))
>>> b
```

```
>>> b = reshape(a,(2,8))
>>> b
array([[ 0.51878337,  0.93337481,  0.84368137,  0.07324918,  0.1
         0.92344357,  0.50366378,  0.59754141],
       [ 0.67841199,  0.73959186,  0.45789404,  0.85003645,  0.9
         0.81794353,  0.78810869,  0.05192744]])
```

Traza y determinante

La traza y el determinante de una matriz, se pueden obtener respectivamente, con la función `trace` y `det`:

```
>>> linalg.det(eye(3))
```

Traza y determinante

La traza y el determinante de una matriz, se pueden obtener respectivamente, con la función `trace` y `det`:

```
>>> linalg.det(eye(3))
1.0
```

Traza y determinante

La traza y el determinante de una matriz, se pueden obtener respectivamente, con la función `trace` y `det`:

```
>>> linalg.det(eye(3))
1.0
>>> trace(eye(3))
```

Traza y determinante

La traza y el determinante de una matriz, se pueden obtener respectivamente, con la función `trace` y `det`:

```
>>> linalg.det(eye(3))
1.0
>>> trace(eye(3))
3.0
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
array([[ 0.64569289,  0.72496086],
       [ 0.98555394,  0.02864243]])
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
array([[ 0.64569289,  0.72496086],
       [ 0.98555394,  0.02864243]])
>>> linalg.inv(a)
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
array([[ 0.64569289,  0.72496086],
       [ 0.98555394,  0.02864243]])
>>> linalg.inv(a)
array([[-0.04115328,  1.04161968],
       [ 1.41603835, -0.92772791]])
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
array([[ 0.64569289,  0.72496086],
       [ 0.98555394,  0.02864243]])
>>> linalg.inv(a)
array([[-0.04115328,  1.04161968],
       [ 1.41603835, -0.92772791]])
>>> dot(a,linalg.inv(a))
```

Inversa de una matriz

La inversa de una matriz se calcula con la función `inv`:

```
>>> a = random.rand(2,2)
>>> a
array([[ 0.64569289,  0.72496086],
       [ 0.98555394,  0.02864243]])
>>> linalg.inv(a)
array([[-0.04115328,  1.04161968],
       [ 1.41603835, -0.92772791]])
>>> dot(a,linalg.inv(a))
array([[ 1.,  0.],
       [ 0.,  1.]])
```

Matriz transpuesta

La transpuesta de una matriz se obtiene con `tranpose`, que puede usarse también como método. Otra manera es usar el atributo `.T`

Como función

```
>>> transpose(a)
```

Matriz transpuesta

La transpuesta de una matriz se obtiene con `tranpose`, que puede usarse también como método. Otra manera es usar el atributo `.T`

Como función

```
>>> transpose(a)
array([[ 0.64569289,  0.98555394],
       [ 0.72496086,  0.02864243]])
```

Como método

```
>>> a.transpose()
```

Como método

```
>>> a.transpose()
array([[ 0.64569289,  0.98555394],
       [ 0.72496086,  0.02864243]])
```

Con el atributo .T

```
>>> a.T
```

Como método

```
>>> a.transpose()
array([[ 0.64569289,  0.98555394],
       [ 0.72496086,  0.02864243]])
```

Con el atributo .T

```
>>> a.T
array([[ 0.64569289,  0.98555394],
       [ 0.72496086,  0.02864243]])
```

Valores y vectores propios

La función `eig` permite obtener los valores y vectores propios:

```
>>> linalg.eig(a)
(array([ 1.23698756, -0.56265224]),
 array([[ 0.77492825, -0.51447164],
        [ 0.63204921,  0.85750739]]))
```

Contenido

Otro ejemplo de la física

El método de Euler que usamos para el problema de la bicicleta, se puede generalizr fácilmente para tratar con el movimiento en dos dimensiones espaciales.

Para ser más específicos, consideramos un proyectil como una bala que se dispara desde un cañón. Tenemos ya inicialmente un gran cañón en la mente, y el cálculo de la distancia máxima que logra la bala es una tarea común en la física.

Si ignoramos la resistencia del aire, las ecuaciones de movimiento, que se obtuvieron de nuevo con la segunda ley de Newton, se puede escribir como:

Ecuaciones de movimiento en dos dimensiones

$$\begin{aligned} \frac{d^2x}{dt^2} &= 0 \\ \frac{d^2y}{dt^2} &= -g \end{aligned}$$

donde x, y son las coordenadas horizontal y vertical respectivamente de la bala, y g es la aceleración debida a la gravedad.

Estas son las ecuaciones diferenciales de segundo orden, a diferencia de las ecuaciones de primer orden que nos hemos encontrado hasta ahora, habrá que generalizar nuestro enfoque.

Hemos visto que con una ecuación de primer orden (con la bicicleta), es posible utilizar una aproximación de diferencias finitas para la derivada y obtener una ecuación algebraica simple, que contiene la variable de interés en dos pasos de tiempo adyacentes.

Sin embargo, si tuviéramos que adoptar el mismo enfoque con una de las ecuaciones de segundo orden y escribir una diferencia finita aproximación a la segunda derivada, obtendríamos una ecuación algebraica más complicada que implicaría evaluar nuestra variable en tres pasos de tiempo.

Sin embargo, en el presente ejemplo, es posible evitar esta complicación si redefinimos las ecuaciones diferenciales de la siguiente manera.

Vamos a escribir cada una de estas ecuaciones de segundo orden en un sistema de dos ecuaciones de primer orden:

$$\begin{aligned}
\frac{dx}{dt} &= v_x \\
\frac{dv_x}{dt} &= 0 \\
\frac{dy}{dt} &= v_y \\
\frac{dv_y}{dt} &= -g
\end{aligned}$$

Ahora tenemos el doble de ecuaciones que en el ejemplo de la bicicleta, pero podemos usar el método de Euler para resolverlas. La aproximación por diferencias finitas queda como:

$$\begin{aligned} x_{i+1} &= x_i + v_{x,i}\Delta t \\ v_{x,i+1} &= v_{x,i} \\ y_{i+1} &= y_i + v_{y,i}\Delta t \\ v_{y,i+1} &= v_{y,i} - g\Delta t \end{aligned}$$

Dados los valores iniciales de x, y, v_x, v_y, podemos usar las ecuaciones anteriores para estimar valores posteriores de tiempo. Aquí tenemos una *aproximación*, dado que los términos de orden $(\Delta)^2$ se cancelan, al elegir un Δt pequeño.

Ejercicio para entregar

Elabora un código en Python para resolver el problema del lanzamiento de una bala de cañon y calcula la distancia máxima de desplazamiento.