```
#!/usr/bin/env python
# coding: utf-8

# In[72]:


#Marlon Deyber Restrepo Rodriguez
#Computación Blanda


# In[5]:


# Se importa la librería numpy

import numpy as np
# APILAMIENTO
# -----------
# Apilado
# Las matrices se pueden apilar horizontalmente, en profundidad o
# verticalmente. Podemos utilizar, para ese propósito,
# las funciones vstack, dstack, hstack, column_stack, row_stack y
concatenate.
# Para empezar, vamos a crear dos arrays
# Matriz a
a = np.arange(20).reshape(4,5)
print('a =\n', a, '\n')
# Matriz b, creada a partir de la matriz a
b = a*5
print('b =\n', b)
# Utilizaremos estas dos matrices para mostrar los mecanismos
# de apilamiento disponibles


# In[6]:


# APILAMIENTO HORIZONTAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento horizontal
print('Apilamiento horizontal =\n', np.hstack((a,b)) )


# In[7]:


# APILAMIENTO HORIZONTAL - Variante
# Utilización de la función: concatenate()
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento horizontal
print( 'Apilamiento horizontal con concatenate = \n',
np.concatenate((b,a), axis=1) )
```

```
# Si axis=1, el apilamiento es horizontal


# In[8]:


# APILAMIENTO VERTICAL
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento vertical =\n', np.vstack((b,a)) )


# In[9]:


# APILAMIENTO VERTICAL - Variante

# Utilización de la función: concatenate()
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento vertical con concatenate =\n',
np.concatenate((b,a), axis=0) )
# Si axis=0, el apilamiento es vertical


# In[10]:


# APILAMIENTO EN PROFUNDIDAD

# En el apilamiento en profundidad, se crean bloques utilizando
# parejas de datos tomados de las dos matrices
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento en profundidad
print( 'Apilamiento en profundidad =\n', np.dstack((b,a)) )


# In[11]:


# APILAMIENTO POR COLUMNAS

# El apilamiento por columnas es similar a hstack()
# Se apilan las columnas, de izquierda a derecha, y tomándolas
# de los bloques definidos en la matriz
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento por columnas =\n',
np.column_stack((b,a)) )
```

```
# In[12]:


# APILAMIENTO POR FILAS

# El apilamiento por fila es similar a vstack()
# Se apilan las filas, de arriba hacia abajo, y tomándolas
# de los bloques definidos en la matriz
# Matrices origen
print('a =\n', a, '\n')
print('b =\n', b, '\n')
# Apilamiento vertical
print( 'Apilamiento por filas =\n',
np.row_stack((b,a)) )


# In[13]:


# DIVISIÓN DE ARRAYS

# Las matrices se pueden dividir vertical, horizontalmente o en
profundidad.
# Las funciones involucradas son hsplit, vsplit, dsplit y split.
# Podemos hacer divisiones de las matrices utilizando su estructura
inicial
# o hacerlo indicando la posición después de la cual debe ocurrir la
división


# In[14]:


a =np.arange(9).reshape(3,3)


# In[15]:


# DIVISIÓN HORIZONTAL
print(a, '\n')
# El código resultante divide una matriz a lo largo de su eje
horizontal
# en tres piezas del mismo tamaño y forma:}
print('Array con división horizontal =\n', np.hsplit(a, 3), '\n')
# El mismo efecto se consigue con split() y utilizando una bandera a 1
print('Array con división horizontal, uso de split() =\n',
np.split(a, 3, axis=1))


# In[16]:


# DIVISIÓN VERTICAL
print(a, '\n')
```

```
# La función vsplit divide el array a lo largo del eje vertical:
print('División Vertical = \n', np.vsplit(a, 3), '\n')
# El mismo efecto se consigue con split() y utilizando una bandera a 0
print('Array con división vertical, uso de split() =\n',
np.split(a, 3, axis=0))


# In[17]:


# DIVISIÓN EN PROFUNDIDAD

# La función dsplit, como era de esperarse, realiza división
# en profundidad dentro del array
# Para ilustrar con un ejemplo, utilizaremos una matriz de rango tres
c = np.arange(27).reshape(3, 3, 3)
print(c, '\n')
# Se realiza la división
print('División en profundidad =\n', np.dsplit(c,3), '\n')


# In[18]:


# PROPIEDADES DE LOS ARRAYS


# In[19]:


# El atributo ndim calcula el número de dimensiones

print(b, '\n')
print('ndim: ', b.ndim)


# In[20]:


# El atributo size calcula el número de elementos

print(b, '\n')
print('size: ', b.size)


# In[21]:


# El atributo itemsize obtiene el número de bytes por cada

# elemento en el array
print('itemsize: ', b.itemsize)


# In[22]:
```

```
# El atributo nbytes calcula el número total de bytes del array

print(b, '\n')
print('nbytes: ', b.nbytes, '\n')
# Es equivalente a la siguiente operación
print('nbytes equivalente: ', b.size * b.itemsize)


# In[23]:


# El atributo T tiene el mismo efecto que la transpuesta de la matriz

b.resize(2,6)
print(b, '\n')
print('Transpuesta: ', b.T)


# In[24]:


# Los números complejos en numpy se representan con j

b = np.array([4.j + 2, 3.j + 6])
print('Complejo: \n', b)


# In[25]:


# El atributo real nos da la parte real del array,
# o el array en sí mismo si solo contiene números reales
print('real: ', b.real, '\n')
# El atributo imag contiene la parte imaginaria del array
print('imaginario: ', b.imag)


# In[26]:


# Si el array contiene números complejos, entonces el tipo de datos

# se convierte automáticamente a complejo
print(b.dtype)


# In[27]:


# El atributo flat devuelve un objeto numpy.flatiter.
# Esta es la única forma de adquirir un flatiter:
# no tenemos acceso a un constructor de flatiter.
# El apartamento El iterador nos permite recorrer una matriz
# como si fuera una matriz plana, como se muestra a continuación:
# En el siguiente ejemplo se clarifica este concepto
b = np.arange(4).reshape(2,2)
print(b, '\n')
```

```
f = b.flat
print(f, '\n')
# Ciclo que itera a lo largo de f
for item in f: print (item)
# Selección de un elemento
print('\n')
print('Elemento 2: ', b.flat[2])
# Operaciones directas con flat
b.flat = 7
print(b, '\n')
b.flat[[1,3]] = 1
print(b, '\n')


# In[ ]:




# In[ ]:
```