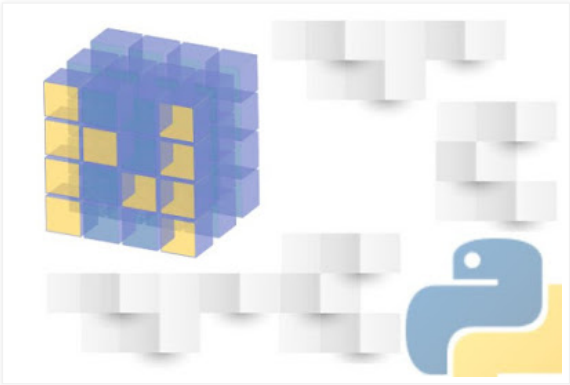
# Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)



**martes, 29 de octubre de 2019**

## Primeros pasos con Numpy



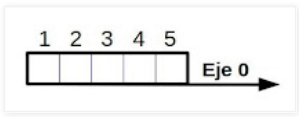
### Introducción

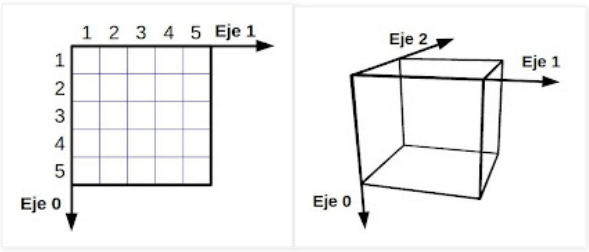
**Numpy** es una biblioteca para Python que facilita el trabajo con arrays (vectores y matrices), un tipo de dato estructurado muy utilizado en análisis de datos, en informática científica y en el área del aprendizaje automático (learning machine).

Numpy permite declarar arrays con distintas dimensiones capaces de albergar gran cantidad de datos del mismo tipo y relacionados entre sí. Además, provee numerosos métodos para manipular los arrays; y para acceder a la información y procesarla de forma muy eficiente.

Básicamente, los datos (elementos) que puede contener un array son cadenas de caracteres, números enteros de distintos tamaños, números reales y booleanos, entre otros.

En un array unidimensional para acceder a dichos datos se utiliza un número (índice) que se corresponde con la posición que ocupa cada elemento en el array. En arrays de más de una dimensión se utilizan tantos indices como dimensiones o ejes existan.




El número de elementos que puede contener como máximo un array define su tamaño; que en arrays multidimensionales viene determinado por el producto del número máximo de elementos en cada eje o longitud de cada dimensión.

**ndarray** es la clase que permite crear objetos array en Numpy. Estos objetos recuerdan a los de la clase array del módulo array de la librería estándar de Python, aunque estos últimos solo permiten operar con arrays unidimensionales, cuentan con menos posibilidades para su procesamiento y tienen un rendimiento inferior.

**Buscar**

Buscar

**Python para impacientes**

Python
IPython
EasyGUI
Tkinter
JupyterLab
Numpy

**Anexos**

Guía urgente de MySQL
Guía rápida de SQLite3

**Entradas + populares**

Dar color a las salidas en la consola
En Python para dar color a las salidas en la consola (o en la terminal de texto) existen varias posibilidades. Hay un método basado ...

Instalación de Python, paso a paso
Instalación de Python 3.6 A finales de 2016 se produjo el lanzamiento de Python 3.6 . El propósito de esta entrada es mostrar, pas...

Variables de control en Tkinter
Variables de control Las variables de control son objetos especiales que se asocian a los widgets para almacenar sus valore...

Añadir, consultar, modificar y suprimir elementos en Numpy
Acceder a los elementos de un array. [], [,], ... Acceder a un elemento de un array. Para acceder a un elemento se utiliz...

Cálculo con arrays Numpy
Numpy ofrece todo lo necesario para obtener un buen rendimiento cuando se trata de hacer cálculos con arrays. Por como está concebid...

Operaciones con fechas y horas. Calendarios
Los módulos datetime y calendar amplían las posibilidades del módulo time que provee funciones para manipular expresiones de ti...

Tkinter: interfaces gráficas en Python
Introducción Con Python hay muchas posibilidades para programar una interfaz gráfica de usuario ( GUI ) pero Tkinter es fácil d...

Convertir, copiar, ordenar, unir y dividir arrays Numpy
Esta entrada trata sobre algunos métodos que se utilizan en Numpy para convertir listas en arrays y viceversa; para copiar arrays d...

Tkinter: Tipos de ventanas
Ventanas de aplicación y de diálogos En la entrada anterior tratamos los distintos gestores de geometría que se utilizan para di...

El módulo random
El módulo random de la librería estándar de Python incluye un conjunto de funciones

## Instalando Numpy

**Instalar Numpy con el gestor de paquetes pip.**

```
$ pip install numpy
```

**Crear un entorno virtual con Numpy y IPython.**

Como suele ser habitual se aconseja crear entornos virtuales para desarrollar cualquier tipo de proyecto Python. Para preparar y utilizar un entorno virtual **venv** con numpy y el entorno interactivo IPython para probar los ejemplos de esta guía para impacientes, seguir los siguientes pasos:

```
1. Crear un entorno virtual para un proyecto:

$ python3 -m venv proyecto1

2. Acceder al directorio del proyecto:

$ cd proyecto1

3. Activar el entorno virtual del proyecto:

$ source bin/activate

4. Instalar IPython y Numpy en el entorno virtual del proyecto:

$ pip install ipython
$ pip install numpy

5. Iniciar IPython para iniciar una sesión de trabajo con Numpy.

$ ipython

6. Terminar una sesión IPython

: quit

7. Desactivar el entorno virtual del proyecto

$ deactivate
```

**Conocer la versión instalada de Numpy.**

```
1. Con pip:

$ pip show numpy

Name: numpy
Version: 1.17.3
Summary: NumPy is the fundamental package for array with Python.
Home-page: https://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD

2. En IPython

: import numpy as np
: np.version.version  # '1.17.3'

: np.__version__  # '1.17.3'
```

También, existen distribuciones Python como Anaconda y ActivePython que incluyen entre sus librerías a Numpy. Estas distribuciones son recomendables para desarrolladores que van a trabajar en proyectos relacionados con el análisis de datos porque proporcionan herramientas como Jupyter, Ipython, etc. y otros paquetes muy usados en este tipo de proyectos como Scipy, Pandas, matplotlib, etc.

**Crear un entorno virtual con Miniconda.**

La distribución ligera Miniconda que deriva de Anaconda es una buena alternativa para comenzar a dar los primeros pasos con Numpy. Para crear un entorno virtual después de su instalación seguir los siguientes pasos:

que permiten obtener de distintos modos números a...

**Archivo**

octubre 2019 (2)

**python.org**

**pypi.org**

**Sitios**

- ActivePython
- Anaconda
- Bpython
- Django
- Flask
- Ipython
- IronPython
- Matplotlib
- MicroPython
- Numpy
- Pandas
- Pillow
- PortablePython
- PyBrain
- PyCharm
- PyDev
- PyGame
- Pypi
- PyPy
- Pyramid
- Python.org
- PyTorch
- SciPy.org
- Spyder
- Tensorflow
- TurboGears

```
1. Crear entorno virtual en Miniconda:

$ conda create -n proyecto1

2. Activar entorno virtual:

$ conda activate proyecto1

3. Instalar Numpy y IPuthon

$ conda install numpy

$ conda install ipython

4. Desactivar entorno virtual:

$ conda deactivate
```

## Declarando arrays con Numpy

**Comenzar una sesión con Numpy.**

Para comenzar una sesión con Numpy, activar el entorno virtual, iniciar IPython e importar el modulo Numpy con el alias **np**:

```
import numpy as np
```

### array()

**Declarar un array.**

```
# Declarar un array con los números enteros del 1 al 5:

a = np.array([1, 2, 3, 4, 5])

# Acceder al objeto array:

a  # array([1, 2, 3, 4, 5])

# Obtener el tipo de objeto declarado:

type(a)  # numpy.ndarray

# Imprimir el array:

print(a)  # [1 2 3 4 5]

# Comprobar si existe un array Numpy llamado a:

if isinstance(a, np.ndarray):
    print('Existe')
else:
    print('No existe')

# Declarar un array bidimensional (2x3) para datos de tipo float:

a = np.array([[1, 2, 3], [4, 5.8, 6]], dtype=float)
print(a)

# [[1.  2.  3. ]
#  [4.  5.8 6. ]]

# Declarar un array tridimensional (2x2x2) para datos de tipo int:

a = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]], dtype=int)
print(a)

# [[[1 2]
#   [3 4]]
#
#  [[5 6]
#   [7 8]]]

# Declarar un array sin datos (vacío):

a = np.array([])
```

```
a  # array([], dtype=float64)
print(a)  # []
```

### arange()

**Declarar un array con una sucesión de valores.**

```
# Declarar un array con 5 valores, del 0 al 4:

a = np.arange(5)
print(a)  # [0 1 2 3 4]

# Declarar un array con valores desde 1 hasta 10 con un
# intervalo de 2:

a = np.arange(1, 10, 2)
print(a)  # [1 3 5 7 9]

# Declarar un array con valores desde 2 hasta 4:

a = np.arange(2, 4)
print(a)  # [2 3]
```

### zeros()

**Declarar un array con todos los elementos con el valor 0.**

```
# Declarar array bidimensional (2x2) con todos los elementos con 0:

a = np.zeros((2, 2))
print(a)

# [[0. 0.]
#  [0. 0.]]
```

### ones()

**Declarar un array con todos los elementos con el valor 1.**

```
# Declarar array bidimensional (3x2) con todos los elementos con 1:

a = np.ones((3, 2))
print(a)

# [[1. 1.]
#  [1. 1.]
#  [1. 1.]]
```

### empty()

**Declarar un array vacío.**

```
# Declarar un array bidimensional (2x2) vacío.

a = np.empty((2, 2))
print(a)

# [[0. 0.]
#  [0. 0.]]

# El array se creará con los datos existentes en las direcciones
# de memoria.
```

### random.randint()

**Declarar un array con números enteros aleatorios.**

```
# Declarar un array (3x3) con enteros aleatorios entre 1 y 9:

a = random.randint(1, 10, (3,3))
print(a)

# [[6 7 2]
#  [1 9 4]
#  [8 2 2]]
```

### random.random()

**Declarar un array con números aleatorios.**

```
# Declarar un array de (3x3) con números aleatorios:

a = np.random.random((3,3))
print(a)

# [[0.46256707 0.05762619 0.84186372]
#  [0.37662455 0.9371823  0.02038264]
#  [0.24177232 0.45393929 0.33990566]]
```

### identity()

**Declarar un array con la matriz identidad.**

Una matriz identidad es una matriz cuadrada donde todos sus elementos son ceros (0) excepto los elementos de la diagonal principal que son unos (1).

```
# Declarar un array (3x3) con la matriz identidad:

a = np.identity(3)
print(a)

# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]
```

### eye()

**Declarar un array con sus elementos con 0 y 1 en la diagonal principal.**

```
# Declarar un array (3x3) con todos sus elementos con 0 y
# 1 en la diagonal principal.

a = np.eye(3)
print(a)

# [[1. 0. 0.]
#  [0. 1. 0.]
#  [0. 0. 1.]]

# Declarar un array (3x3) con todos sus elementos con 0 y 1 en
# la diagonal 1.

# k es un índice que representa la diagonal en la que los elementos
# tendrán el valor 1: si k es 0 se trata de la diagonal principal;
# si es un número positivo una diagonal superior y si es un número
# negativo una inferior.

a = np.eye(3, k=1)
print(a)

# [[0. 1. 0.]
#  [0. 0. 1.]
#  [0. 0. 0.]]

# Declarar un array (3x5) con todos sus elementos con 0 y 1 en la
# diagonal -2.

a = np.eye(3, 5, k=-2)
print(a)
```

```
# [[0. 0. 0. 0. 0.]
#  [0. 0. 0. 0. 0.]
#  [1. 0. 0. 0. 0.]]
```

### full()

**Declarar un array con todos los elementos con el mismo valor.**

```
# Declarar un array (3x3) con todos los elementos con 5:

a = np.full((3, 3), 5)
print(a)

# [[5 5 5]
#  [5 5 5]
#  [5 5 5]]
```

### asarray()

**Declarar un array con los elementos de una lista o una tupla.**

```
# Declarar un array con los elementos de una tupla:

tupla = (1, 2, 3, 4, 5)
a = np.asarray(tupla)
print(a)  # [1 2 3 4 5]

# Declarar un array (2x3) con los elementos de una lista:

lista = [[1, 2, 3], [4, 5, 6]]
a = np.asarray(lista)
print(a)

# [[1 2 3]
#  [4 5 6]]
```

### full_like()

**Declarar un array con dimensiones y forma iguales que otro array.**

```
# Declarar un array con dimensiones y forma de otro array pero
# con otros valores:

matriz = np.array([(1, 2, 3), (4, 5, 6)])
a = np.full_like(matriz, -1)
print(a)

# [[-1 -1 -1]
#  [-1 -1 -1]]

Otras opciones: zeros_like() y ones_like()
```

### frombuffer()

**Declarar un array con enteros a partir de una cadena de bytes.**

```
# Declarar un array con 12 enteros a partir de una cadena de bytes:

cadena = b'Python para impacientes'
a = np.frombuffer(cadena, count=12, dtype=np.int8)
print(a)  # [ 80 121 116 104 111 110  32 112  97 114  97  32]

# Declarar un array de cadenas de 1 byte a partir de una cadena.

cadena = b'Python para impacientes'
a = np.frombuffer(cadena, count=6, dtype='S1')
print(a)  # [b'P' b'y' b't' b'h' b'o' b'n']
```

### linspace()

**Declarar un array con valores equidistantes.**

```
# Declarar un array de 5 elementos con valores entre 0 y 2:

a = np.linspace(0, 2, num=5)
print(a)  #  [0.  0.5 1.  1.5 2. ]
```

### fromfunction()

**Declarar un array a partir de los datos de una función.**

```
# Declarar array a partir de los datos de una función lambda:

a = np.fromfunction(lambda x: x*x, (10,), dtype=int)
print(a)  # [ 0  1  4  9 16 25 36 49 64 81]

# Declarar un array (5x5) a partir de los datos de una función.

a = np.fromfunction(lambda x, y: x-y-1, (5, 5), dtype=int)
print(a)

# [[-1 -2 -3 -4 -5]
#  [ 0 -1 -2 -3 -4]
#  [ 1  0 -1 -2 -3]
#  [ 2  1  0 -1 -2]
#  [ 3  2  1  0 -1]]
```

### fromiter()

**Declarar un array con los valores devueltos por un iterador.**

```
# Declarar vector con valores decrecientes devueltos por iterador:

iterador = [v for v in range(100,1,-3)]
a = np.fromiter(iterador, int, count=10)
print(a)  # [100  97  94  91  88  85  82  79  76  73]
```

### fromstring()

**Declarar un array a partir de una cadena de texto.**

```
# Declarar vector con booleanos a partir los valores de una cadena:

a = np.fromstring('0,0,0,1,1,0,1', dtype=bool, sep=',')  #
print(a)  # [False False False  True  True False  True]
```

### diag()

**Declarar un array a partir de la diagonal de otro array.**

```
# Declarar un array a partir de la diagonal principal de otro:

a = np.array([(1, 5, 10), (5, 10, 15), (10, 15, 20)])
diagonal = np.diag(a)  # array([ 1, 10, 20])
b = np.diag(diagonal)
print(b)

# [[ 1  0  0]
#  [ 0 10  0]
#  [ 0  0 20]]

# Declarar un array a partir de elementos de la diagonal -1:

a = np.array([(1, 5, 10), (5, 10, 15), (10, 15, 20)])
diagonal = np.diag(a, k=-1)  # array([ 5, 15])
b = np.diag(diagonal)
print(b)
```

```
# [[ 5  0]
#  [ 0 15]]
```

### diagflat()

**Declarar un array con los elementos de otro en la diagonal.**

```
a = np.array([(0, 5), (5, 10)])
b = np.diagflat(a)
print(b)

# [[ 0  0  0  0]
#  [ 0  5  0  0]
#  [ 0  0  5  0]
#  [ 0  0  0 10]]
```

### tri()

**Declarar array con diagonal principal e inferiores a 1 y resto a 0.**

```
a = np.tri(4)
print(a)

# [[1. 0. 0. 0.]
#  [1. 1. 0. 0.]
#  [1. 1. 1. 0.]
#  [1. 1. 1. 1.]]
```

### dtype={'names':(),'formats':()})

**Declarar arrays con datos estructurados a partir de listas.**

```
# Declarar listas con nombres y edades.

# Los datos de ambas listas están relacionados por su posición:
# la edad de María es 24 años, la de Carlos es 25, etc.

nombre = ['Maria', 'Carlos', 'Ana', 'Luisa']
edad = [24, 25, 35, 19]

# Declarar array vacío con dos campos:

datos = np.empty(4, dtype={'names':('nombre', 'edad'),
                           'formats':('U15', 'i4')})

# Asignar los datos de las listas a los campos del array:

datos['nombre'] = nombre
datos['edad'] = edad

# Obtener todos los nombres:

print(datos['nombre'])  # ['Maria' 'Carlos' 'Ana' 'Luisa']

# Obtener todas las edades:

print(datos['edad'])  # [24 25 35 19]

# Obtener los datos de la primera persona:

print(datos[0])  # ('Maria', 24)

# Obtener los datos de la última persona:

print(datos[-1])  # ('Luisa', 19)

# Obtener la primera edad:

print(datos[0]['edad'])  # 24

# Obtener array con todos los nombres con más de 24 años:
```

```
print(datos[datos['edad'] > 24]['nombre'])  # ['Carlos' 'Ana']
```

### core.records.fromarrays()

**Declarar un registro de arrays.**

Permite operar con los elementos de varios arrays de distintos tipos agrupados en un nuevo tipo de objeto llamado **registro** que funciona como un array.

```
a = np.array(['a', 'b', 'c', 'd', 'e'])
b = np.array([1, 2, 3, 4, 5])
registro = np.core.records.fromarrays([a, b],names='a, b')
print(registro.a)  # ['a' 'b' 'c' 'd' 'e']
print(registro.b)  # [1 2 3 4 5]
print(registro[0])  # ('a', 1)
registro[2] = ('x', 0)
print(registro.a)  # ['a' 'b' 'x' 'd' 'e']
print(registro)  # [('a', 1) ('b', 2) ('x', 0) ('d', 4) ('e', 5)]
```

La próxima entrada la dedicamos a los tipos de arrays que existen atendiendo a los datos que pueden contener y a las formas que existen para definirlos.

Publicado por Pherkad en 3:50

Etiquetas: Numpy

Entrada más reciente Inicio Entrada antigua