Tema 0 - Programación básica con Python III

Curso de Física Computacional

M. en C. Gustavo Contreras Mayén

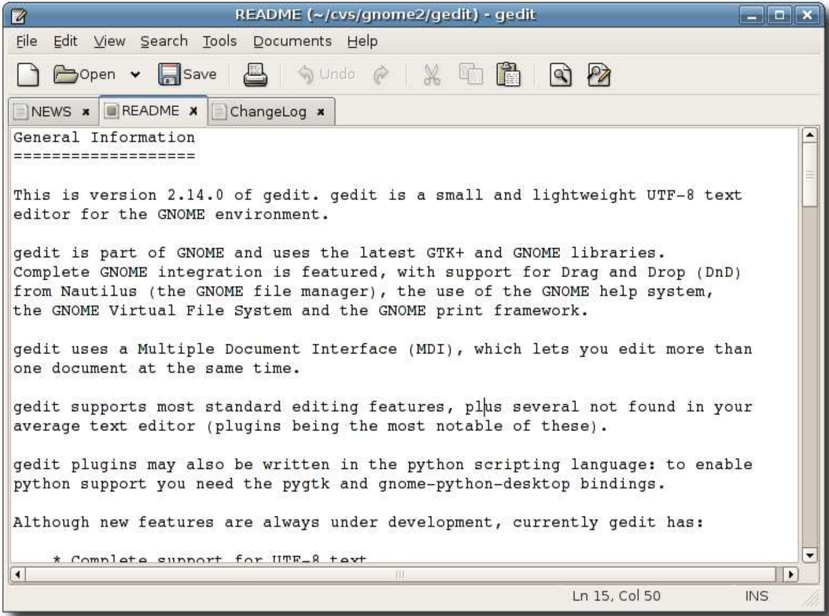
Contenido

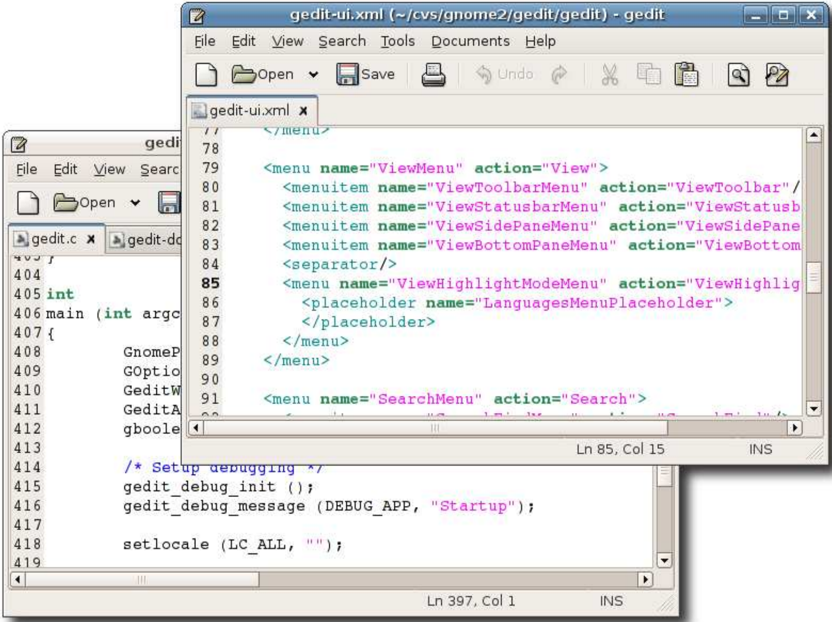
Usando gEdit

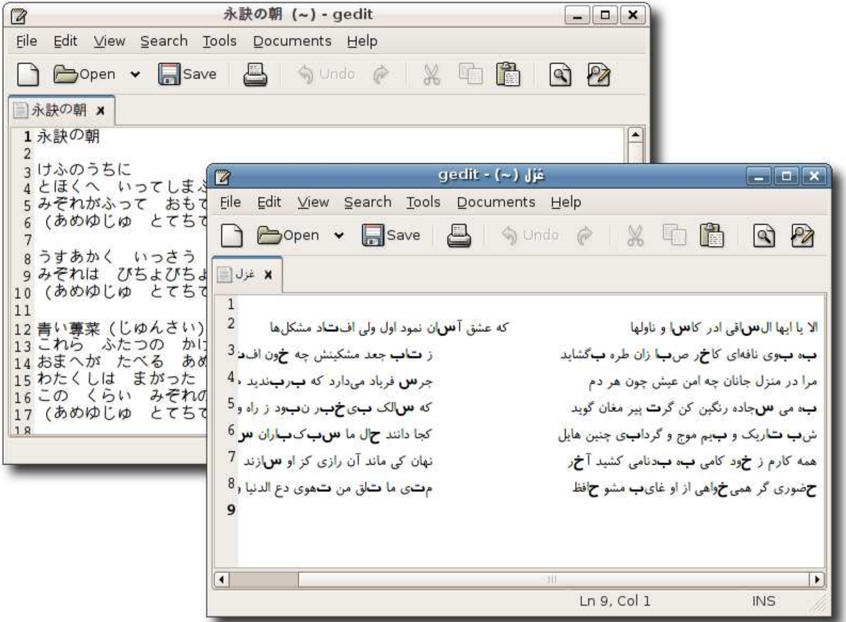
gedit es un editor de textos compatible con UTF-8 para GNU/Linux, Mac OS X y Microsoft Windows. Diseñado como un editor de textos de propósito general, gedit enfatiza la simplicidad y facilidad de uso. Incluye herramientas para la edición de código fuente y textos estructurados, como lenguajes de marcado.

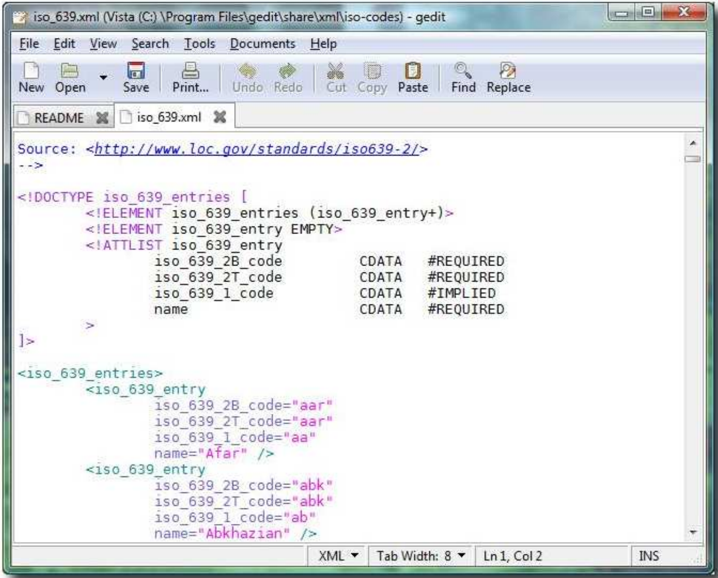
Es el editor predeterminado de GNOME.

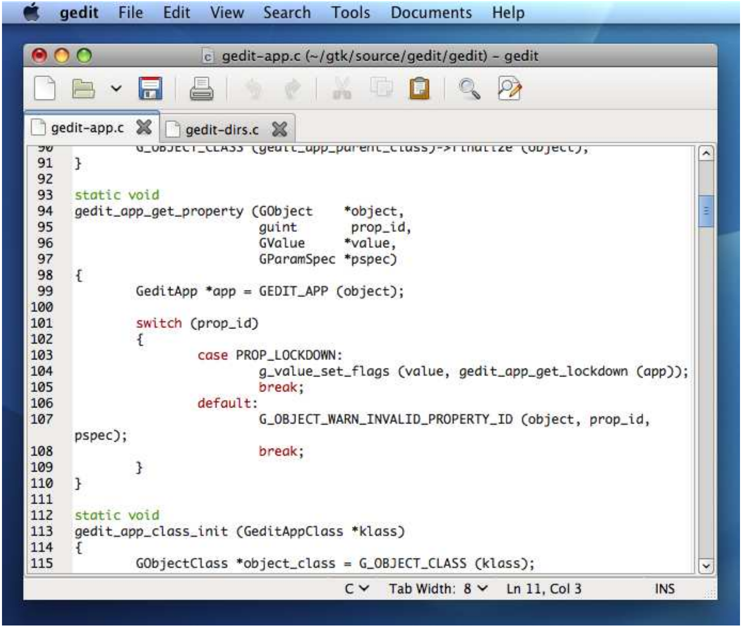
Distribuido bajo las condiciones de la licencia GPL, gedit es software libre.


README (~/cvs/gnome2/gedit) - gedit
File Edit View Search Tools Documents Help
Open Save Undo
NEWS README ChangeLog
General Information
===================
This is version 2.14.0 of gedit. gedit is a small and lightweight UTF-8 text editor for the GNOME environment.
gedit is part of GNOME and uses the latest GTK+ and GNOME libraries. Complete GNOME integration is featured, with support for Drag and Drop (DnD) from Nautilus (the GNOME file manager), the use of the GNOME help system, the GNOME Virtual File System and the GNOME print framework.
gedit uses a Multiple Document Interface (MDI), which lets you edit more than one document at the same time.
gedit supports most standard editing features, plus several not found in your average text editor (plugins being the most notable of these).
gedit plugins may also be written in the python scripting language: to enable python support you need the pygtk and gnome-python-desktop bindings.
Although new features are always under development, currently gedit has:
Ln 15, Col 50
INS


gedit-ui.xml (~/cvs/gnome2/gedit/gedit) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
gedit-ui.xml
<menu name="ViewMenu" action="View">
<menuitem name="ViewToolbarMenu" action="ViewToolbar"/
<menuitem name="ViewStatusbarMenu" action="ViewStatusb
<menuitem name="ViewSidePaneMenu" action="ViewSidePane
<menuitem name="ViewBottomPaneMenu" action="ViewBottom
<separator/>
<menu name="ViewHighlightModeMenu" action="ViewHighlig
<placeholder name="LanguagesMenuPlaceholder">
</placeholder>
</menu>
</menu>
<menu name="SearchMenu" action="Search">
Ln 85, Col 15
INS
File Edit View Searc
Open
gedit.c
int
main (int argc
{
GnomeP
GOptio
GeditW
GeditA
gboole
/* Setup debugging */
gedit_debug_init ();
gedit_debug_message (DEBUG_APP, "Startup");
setlocale (LC_ALL, "");
Ln 397, Col 1
INS


永訣の朝 (~) - gedit
File Edit View Search Tools Documents Help
Open
Save
Undo
永訣の朝
gedit - (~) غزل
File Edit View Search Tools Documents Help
Open
Save
Undo
غزل
Ln 9, Col 1
INS


iso_639.xml (Vista (C:) \Program Files\gedit\share\xml\iso-codes) - gedit
File Edit View Search Tools Documents Help
New Open Save Print... Undo Redo Cut Copy Paste Find Replace
README iso_639.xml
Source: <http://www.loc.gov/standards/iso639-2/>
-->
<!DOCTYPE iso_639_entries [
<!ELEMENT iso_639_entries (iso_639_entry+)>
<!ELEMENT iso_639_entry EMPTY>
<!ATTLIST iso_639_entry
iso_639_2B_code CDATA #REQUIRED
iso_639_2T_code CDATA #REQUIRED
iso_639_1_code CDATA #IMPLIED
name CDATA #REQUIRED
>
]>
<iso_639_entries>
<iso_639_entry
iso_639_2B_code="aar"
iso_639_2T_code="aar"
iso_639_1_code="aa"
name="Afar" />
<iso_639_entry
iso_639_2B_code="abk"
iso_639_2T_code="abk"
iso_639_1_code="ab"
name="Abkhazian" />
XML Tab Width: 8 Ln 1, Col 2 INS


gedit File Edit View Search Tools Documents Help
gedit-app.c (~/gtk/source/gedit/gedit) - gedit
gedit-app.c
gedit-dirs.c
}
static void
gedit_app_get_property (GObject *object,
guint prop_id,
GValue *value,
GParamSpec *pspec)
{
GeditApp *app = GEDIT_APP (object);
switch (prop_id)
{
case PROP_LOCKDOWN:
g_value_set_flags (value, gedit_app_get_lockdown (app));
break;
default:
G_OBJECT_WARN_INVALID_PROPERTY_ID (object, prop_id,
pspec);
break;
}
}
static void
gedit_app_class_init (GeditAppClass *klass)
{
GObjectClass *object_class = G_OBJECT_CLASS (klass);
C
Tab Width: 8
Ln 11, Col 3
INS

Contenido

El entorno Spyder2

Spyder es un entorno de desarrollo integrado para el lenguaje Python con pruebas interactivas y avanzadas funciones de depuración, introspección y edición.

Spyder permite trabajar fácilmente con las mejores herramientas de la pila científica de Python en un entorno sencillo y potente.

Estas son algunas de las características clave de Spyder:

1. Cuadro de diálogo de administración de `PYTHONPATH` como de MATLAB (funciona con todas las consolas)

Estas son algunas de las características clave de Spyder:

1. Cuadro de diálogo de administración de `PYTHONPATH` como de MATLAB (funciona con todas las consolas)
2. Editor de variables de entorno de usuario actual.

Estas son algunas de las características clave de Spyder:

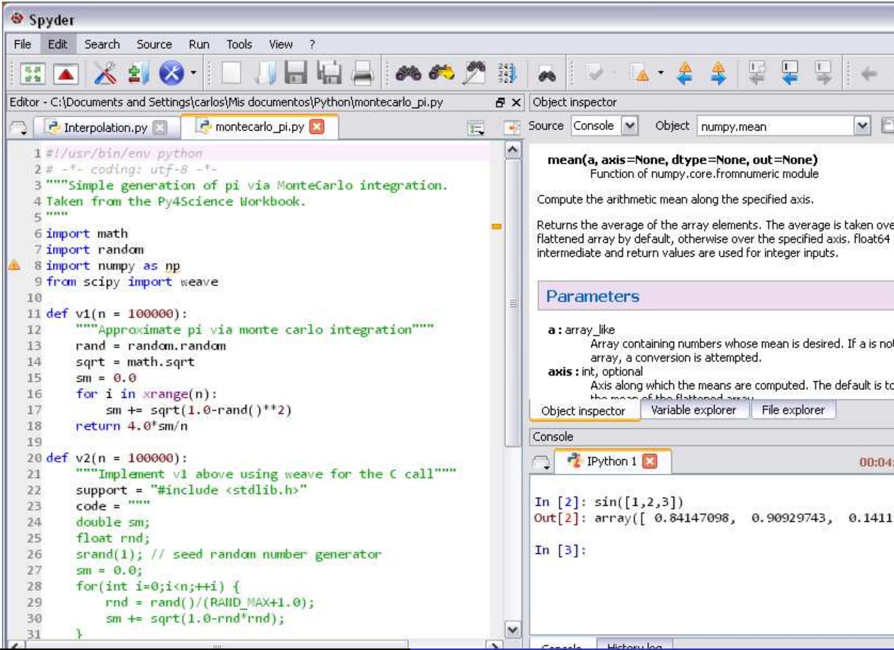
1. Cuadro de diálogo de administración de `PYTHONPATH` como de MATLAB (funciona con todas las consolas)
2. Editor de variables de entorno de usuario actual.
3. Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)

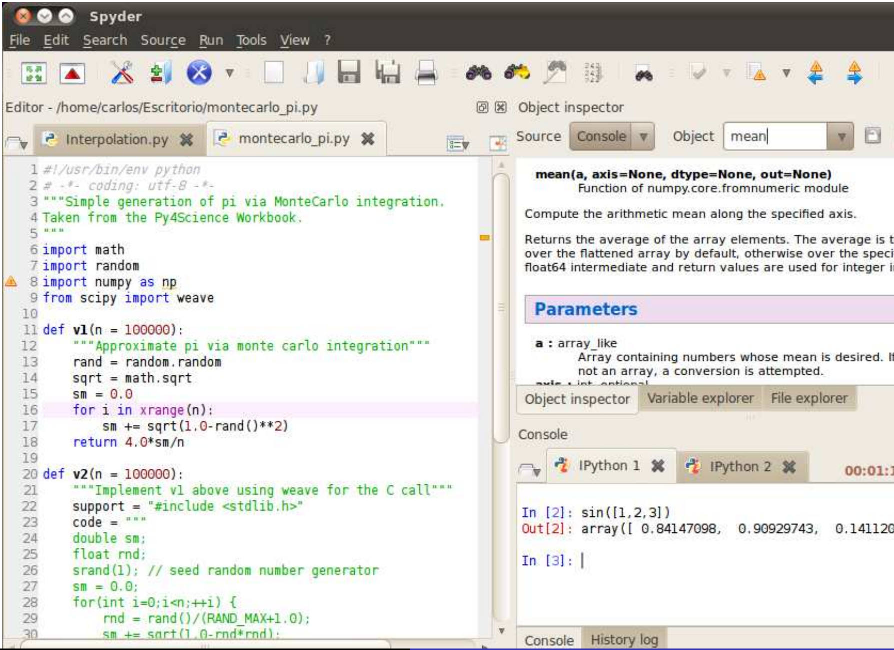
Estas son algunas de las características clave de Spyder:

1. Cuadro de diálogo de administración de `PYTHONPATH` como de MATLAB (funciona con todas las consolas)
2. Editor de variables de entorno de usuario actual.
3. Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)
4. Enlace directo al lanzador de Python(x,y)

Estas son algunas de las características clave de Spyder:

1. Cuadro de diálogo de administración de `PYTHONPATH` como de MATLAB (funciona con todas las consolas)
2. Editor de variables de entorno de usuario actual.
3. Enlaces directos a la documentación (Python, Matplotlib, NumPy, Spicy, etc.)
4. Enlace directo al lanzador de Python(x,y)
5. Enlaces directos a QtDesigner, QtLinguist y QtAssistant (documentación de Qt)


Spyder
File Edit Search Source Run Tools View ?
Editor - C:\Documents and Settings\carlos\Mis documentos\Python\montecarlo_pi.py
Interpolation.py
montecarlo_pi.py
#!/usr/bin/env python
-*- coding: utf-8 -*-
"""Simple generation of pi via MonteCarlo integration.
Taken from the Py4Science Workbook.
"""
import math
import random
import numpy as np
from scipy import weave
def v1(n = 100000):
"""Approximate pi via monte carlo integration"""
rand = random.random
sqrt = math.sqrt
sm = 0.0
for i in xrange(n):
sm += sqrt(1.0-rand()**2)
return 4.0*sm/n
def v2(n = 100000):
"""Implement v1 above using weave for the C call"""
support = "#include <stdlib.h>"
code = """
double sm;
float rnd;
srand(1); // seed random number generator
sm = 0.0;
for(int i=0;i<n;++i) {
rnd = rand()/(RAND_MAX+1.0);
sm += sqrt(1.0-rnd*rnd);
}
Object inspector
Source Console Object numpy.mean
mean(a, axis=None, dtype=None, out=None)
Function of numpy.core.fromnumeric module
Compute the arithmetic mean along the specified axis.
Returns the average of the array elements. The average is taken ove
flattened array by default, otherwise over the specified axis. float64
intermediate and return values are used for integer inputs.
Parameters
a : array_like
Array containing numbers whose mean is desired. If a is no
array, a conversion is attempted.
axis : int, optional
Axis along which the means are computed. The default is to
Object inspector
Variable explorer
File explorer
Console
IPython 1
In [2]: sin([1,2,3])
Out[2]: array([0.84147098, 0.90929743, 0.1411
In [3]:


Spyder
File Edit Search Source Run Tools View ?
Editor - /home/carlos/Escritorio/montecarlo_pi.py
Interpolation.py
montecarlo_pi.py
#!/usr/bin/env python
-*- coding: utf-8 -*-
"""Simple generation of pi via MonteCarlo integration.
Taken from the Py4Science Workbook.
"""
import math
import random
import numpy as np
from scipy import weave
def v1(n = 100000):
"""Approximate pi via monte carlo integration"""
rand = random.random
sqrt = math.sqrt
sm = 0.0
for i in xrange(n):
sm += sqrt(1.0-rand()**2)
return 4.0*sm/n
def v2(n = 100000):
"""Implement v1 above using weave for the C call"""
support = "#include <stdlib.h>"
code = """
double sm;
float rnd;
srand(1); // seed random number generator
sm = 0.0;
for(int i=0;i<n;++i) {
rnd = rand()/(RAND_MAX+1.0);
Object inspector
Source
Console
Object
mean
mean(a, axis=None, dtype=None, out=None)
Function of numpy.core.fromnumeric module
Compute the arithmetic mean along the specified axis.
Parameters
a : array_like
Array containing numbers whose mean is desired.
not an array, a conversion is attempted.
Object inspector
Variable explorer
File explorer
Console
IPython 1
IPython 2
In [2]: sin([1,2,3])
Out[2]: array([0.84147098, 0.90929743,
In [3]:
Console
History log

Contenido

Graficación con Python

Una buena parte del trabajo que tendremos que hacer como físicos es utilizar un conjunto de datos que por si solos, no van a darnos información sobre un modelo o un fenómeno, por ello, será necesario usar gráficas.

Python incluye un módulo de graficación bastante versátil para generar gráficas y exportarlas a diferentes tipos de archivos.

La librería se llama `matplotlib`. Haremos algunos ejercicios para demostrar su potencia.

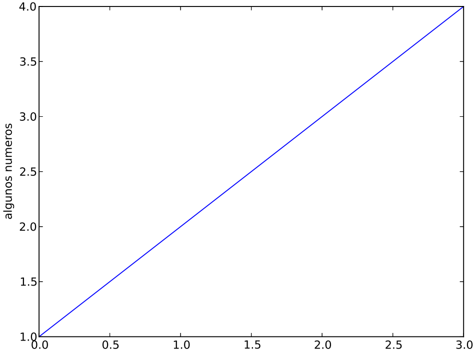
`matplotlib.pyplot` es una colección de funciones de estilo de mando, de tal manera que matplotlib funciona a la manera de MATLAB. Cada instrucción `pyplot` aplica un cambio a una figura: por ejemplo, crear una figura, crear un área de trazado en una figura, trazar algunas líneas en un área de trazado, decorar con etiquetas, etc.

Ejecicio 1

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('algunos numeros')
plt.show()
```

Ejecicio 1

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4])
plt.ylabel('algunos numeros')
plt.show()
```



Te estarás preguntando por qué tenemos en el eje x el rango $0 - 3$ y en el eje y el rango $1 - 4$.

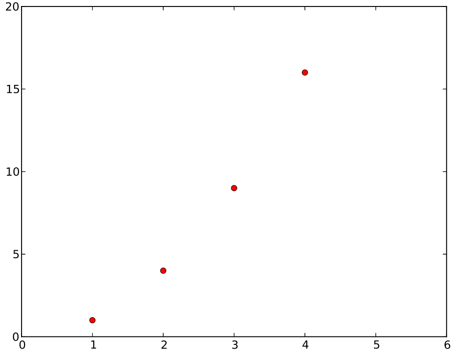
Si proporcionamos una única lista o matriz en el comando `plot`, `matplotlib` asume que es una secuencia de valores de y, por lo que genera automáticamente los valores de x para nosotros. Como los índices en Python comienzan en 0, el vector x por defecto tiene la misma longitud que y, pero inicia con 0. De ahí que los datos x son $[0, 1, 2, 3]$.

Ejecicio 2

```
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```

Ejecicio 2

```python
import matplotlib.pyplot as plt
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
plt.show()
```



Por cada par x, y de argumentos, existe un tercer argumento opcional, que es la cadena de formato que indica el color y tipo de línea.

Las letras y los símbolos de la cadena de formato son como en MATLAB, y concatenar una cadena de color con una cadena estilo de línea.

La cadena de formato por defecto es `'b-'`, que es una línea de color azul.

carácter	descripción
'-'	línea sólida
'--'	línea cortada
'-.'	línea-punto
':'	línea de puntos
'.'	marca de punto
','	marca de pixel
'o'	marca de círculo
'v'	marca de triándulo hacia abajo
'^'	marca de triángulo hacia arriba

carácter	color
'b'	azul
'g'	verde
'r'	rojo
'c'	cyan
'm'	magenta
'y'	amarillo
'k'	negro
'w'	blanco

`matplotlib` se limita a trabajar con listas, por lo que sería bastante acotado para el procesamiento y análisis numérico.

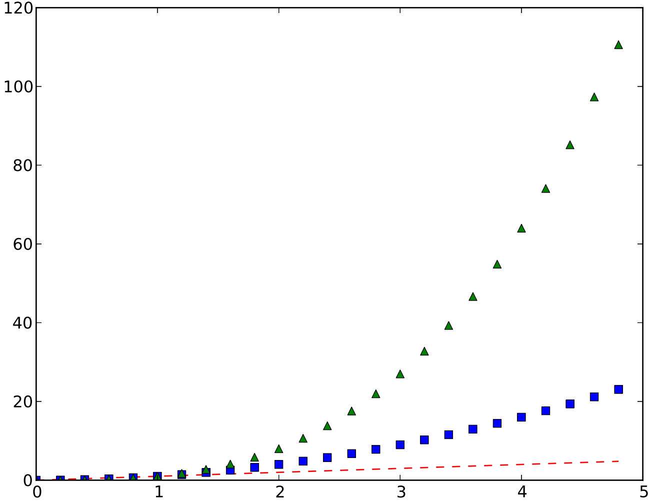
Por lo general, se utilizan los arreglos del módulo `numpy`. De hecho, todas las secuencias se convierten en matrices de `numpy` internamente.

El siguiente ejemplo ilustra un trazado de líneas con varios estilos diferentes en una sola instucción utilizando arreglos.

Ejecicio 3

```python
import numpy as np
import matplotlib.pyplot as plt

t = np.arange(0., 5., 0.2)
plt.plot(t, t, 'r--', t, t**2, 'bs', t, t**3,
    'g^')
plt.show()
```


120
100
80
60
40
20
0
0
1
2
3
4
5

Ejercicio 4

Trabajando con múltiples gráficas

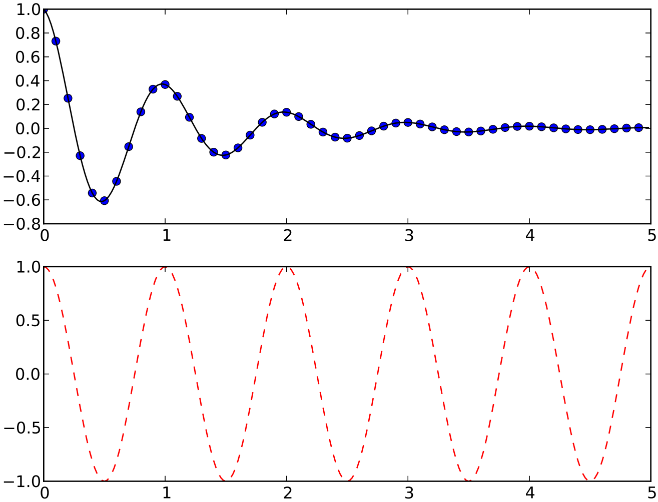
```python
import numpy as np
import matplotlib.pyplot as plt

def f(t):
    return np.exp(-t) * np.cos(2*np.pi*t)

t1 = np.arange(0.0, 5.0, 0.1)
t2 = np.arange(0.0, 5.0, 0.02)

plt.figure(1)
plt.subplot(211)
plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')

plt.subplot(212)
plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```


1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
0
1
2
3
4
5
1.0
0.5
0.0
−0.5
−1.0
0
1
2
3
4
5

El comando `figure()` aquí es opcional, ya `figure(1)` se crea de forma predeterminada, así mismo `subplot(111)` se crea de forma predeterminada si no se especifica manualmente un eje.

El comando `subplot()` especifica `numrows, numcols, fignum` donde `fignum` varía en rango de 1 a `numrows * numcols`. Las comas en el comando `subplot()` son opcionales si `numrows * numcols` < 10. Por tanto `subplot(211)` es idéntica a la `subplot(2,1,1)`.

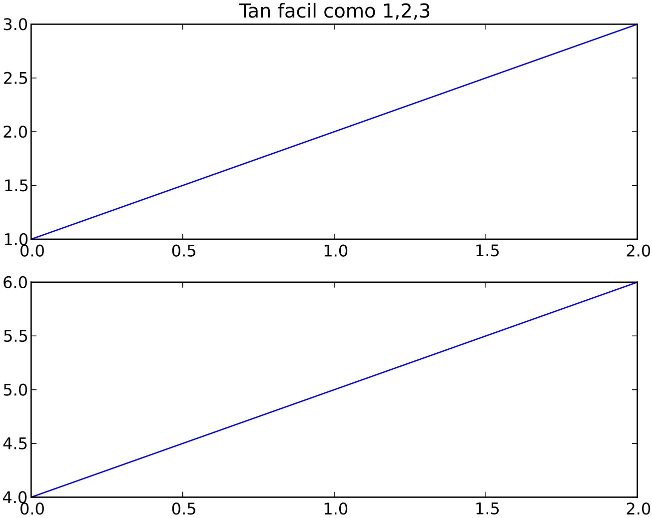
Ejercicio 5

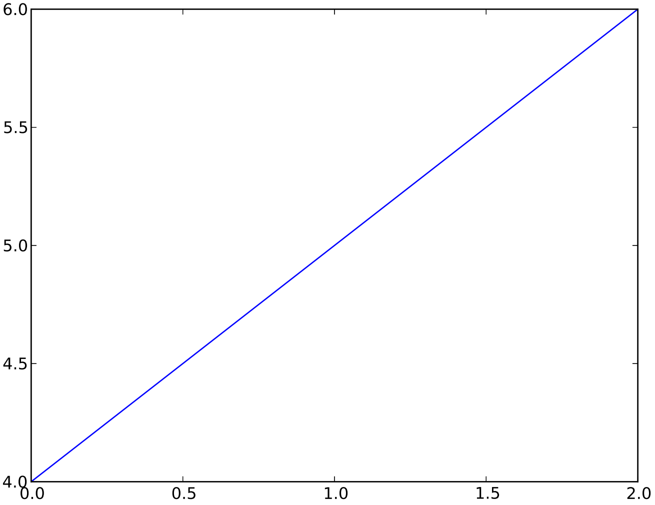
```python
import matplotlib.pyplot as plt

plt.figure(1)
plt.subplot(211)
plt.plot([1,2,3])
plt.subplot(212)
plt.plot([4,5,6])


plt.figure(2)
plt.plot([4,5,6])

plt.figure(1)
plt.subplot(211)
plt.title('Tan facil como 1,2,3')
plt.show()
```


Tan facil como 1,2,3
3.0
2.5
2.0
1.5
1.0
0.0
0.5
1.0
1.5
2.0
6.0
5.5
5.0
4.5
4.0
0.0
0.5
1.0
1.5
2.0


6.0
5.5
5.0
4.5
4.0
0.0
0.5
1.0
1.5
2.0

Contenido

4. Arreglos
   - Crear arreglos
   - Operaciones con arreglos
   - Funciones sobre arreglos
   - Arreglos aleatorios
   - Obtener elementos de un arreglo
   - Algunos métodos convenientes
   - Productos entre arreglos
     - Producto interno (vector-vector)
     - Producto matriz-vector
     - Producto matriz-matriz

Repaso express sobre arreglos

Las estructuras de datos que hemos visto hasta ahora permiten manipular datos de manera muy flexible. Combinándolas y anidándolas, es posible organizar información de manera estructurada para representar sistemas del mundo real.

En muchas aplicaciones en Ciencias, más importante que la organización de los datos es la capacidad de hacer muchas operaciones a la vez sobre grandes conjuntos de datos numéricos de manera eficiente. Algunos ejemplos de problemas que requieren manipular grandes secuencias de números son: la predicción del clima, simulación, graficación de modelos, la construcción de edificios, y el análisis de indicadores financieros entre muchos otros.

La estructura de datos que sirve para almacenar estas grandes secuencias de números (generalmente de tipo float) es el **arreglo**.

Los arreglos tienen algunas similitudes con las listas:

- los elementos tienen un orden y se pueden acceder mediante su posición.
- los elementos se pueden recorrer usando un ciclo `for`.

Sin embargo, también tienen algunas restricciones:

- todos los elementos del arreglo deben tener el mismo tipo.
- en general, el tamaño del arreglo es fijo (no van creciendo dinámicamente como las listas).
- se ocupan principalmente para almacenar datos numéricos.

Los arreglos son los equivalentes en programación de las matrices y vectores en matemáticas. Precisamente, una gran motivación para usar arreglos es que hay mucha teoría detrás de ellos que puede ser usada en el diseño de algoritmos para resolver problemas verdaderamente interesantes.

Crear arreglos

El módulo que provee las estructuras de datos y las funciones para trabajar con arreglos es `NumPy`.

```
from numpy import array
```

Como estaremos usando frecuentemente muchas funciones de este módulo, conviene importarlas todas de una vez usando la siguiente sentencia:

```
from numpy import *
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

```
>>> b = array([6.0, 1, 3, 9, 8])
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

```
>>> b = array([6.0, 1, 3, 9, 8])
>>> b
```

El tipo de datos de los arreglos se llama array. Para crear un arreglo nuevo, se puede usar la función array pasándole como parámetro la lista de valores que deseamos agregar al arreglo:

```
>>> a = array([6, 1, 3, 9, 8])
>>> a
array([6, 1, 3, 9, 8])
```

Todos los elementos del arreglo tienen exactamente el mismo tipo. Para crear un arreglo de números reales, basta con que uno de los valores lo sea:

```
>>> b = array([6.0, 1, 3, 9, 8])
>>> b
array([ 6.,  1.,  3.,  9.,  8.])
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
array([ 6.,  1.,  3.,  9.,  8.])
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
array([ 6.,  1.,  3.,  9.,  8.])
>>> a.astype(complex)
```

Otra opción es convertir el arreglo a otro tipo usando el método `astype`:

```
>>> a
array([6, 1, 3, 9, 8])
>>> a.astype(float)
array([ 6.,  1.,  3.,  9.,  8.])
>>> a.astype(complex)
array([ 6.+0.j,  1.+0.j,  3.+0.j,  9.+0.j,  8
```

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- `zeros(n)` crea un arreglo de *n* ceros.

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- `zeros(n)` crea un arreglo de n ceros.
- `ones(n)` crea un arreglo de n unos.

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- `zeros(n)` crea un arreglo de n ceros.
- `ones(n)` crea un arreglo de n unos.
- `arange(a, b, c)` crea un arreglo de forma similar a la función `range`, con las diferencias que a, b y c pueden ser reales, y que el resultado es un arreglo y no una lista.

Hay muchas formas de arreglos que aparecen a menudo en la práctica, por lo que existen funciones especiales para crearlos:

- `zeros(n)` crea un arreglo de n ceros.
- `ones(n)` crea un arreglo de n unos.
- `arange(a, b, c)` crea un arreglo de forma similar a la función `range`, con las diferencias que a, b y c pueden ser reales, y que el resultado es un arreglo y no una lista.
- `linspace(a, b, n)` crea un arreglo de n valores equiespaciados entre a y b.

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> arange(3.0, 9.0)
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> arange(3.0, 9.0)
array([ 3.,  4.,  5.,  6.,  7.,  8.])
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> arange(3.0, 9.0)
array([ 3.,  4.,  5.,  6.,  7.,  8.])
>>> linspace(1, 2, 5)
```

```
>>> zeros(6)
array([ 0.,  0.,  0.,  0.,  0.,  0.])
>>> ones(5)
array([ 1.,  1.,  1.,  1.,  1.])
>>> arange(3.0, 9.0)
array([ 3.,  4.,  5.,  6.,  7.,  8.])
>>> linspace(1, 2, 5)
array([ 1.  ,  1.25,  1.5 ,  1.75,  2.  ])
```

Operaciones con arreglos

Las limitaciones que tienen los arreglos respecto de las listas son compensadas por la cantidad de operaciones convenientes que permiten realizar sobre ellos.

Las operaciones aritméticas entre arreglos se aplican elemento a elemento:

```
>>> a = array([55, 21, 19, 11,  9])
>>> b = array([12, -9,  0, 22, -9])
```

Operaciones con arreglos

Las limitaciones que tienen los arreglos respecto de las listas son compensadas por la cantidad de operaciones convenientes que permiten realizar sobre ellos.

Las operaciones aritméticas entre arreglos se aplican elemento a elemento:

```
>>> a = array([55, 21, 19, 11,  9])
>>> b = array([12, -9,  0, 22, -9])
```

sumar los dos arreglos elemento a elemento

```
>>> a + b
array([67, 12, 19, 33,  0])
```

multiplicar por 0.1 todos los elementos

```
>>> 0.1 * a
array([ 5.5,  2.1,  1.9,  1.1,  0.9])
```

Note que si quisiéramos hacer estas operaciones usando listas, necesitaríamos usar un ciclo para hacer las operaciones elemento a elemento.

multiplicar por 0.1 todos los elementos

```
>>> 0.1 * a
array([ 5.5,  2.1,  1.9,  1.1,  0.9])
```

restar 9.0 a todos los elementos

```
>>> a - 9.0
array([ 46.,  12.,  10.,   2.,   0.])
```

Note que si quisiéramos hacer estas operaciones usando listas, necesitaríamos usar un ciclo para hacer las operaciones elemento a elemento.

Las operaciones relacionales también se aplican elemento a elemento, y devuelven un arreglo de valores booleanos:

Las operaciones relacionales también se aplican elemento a elemento, y devuelven un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1])/10
>>> a < b
```

Las operaciones relacionales también se aplican elemento a elemento, y devuelven un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1])/10
>>> a < b
array([False,  True,  True, False], dtype=bool)
```

Las operaciones relacionales también se aplican elemento a elemento, y devuelven un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1])/10
>>> a < b
array([False,  True,  True, False], dtype=bool)
>>> a == c
```

Las operaciones relacionales también se aplican elemento a elemento, y devuelven un arreglo de valores booleanos:

```
>>> a = array([5.1, 2.4, 3.8, 3.9])
>>> b = array([4.2, 8.7, 3.9, 0.3])
>>> c = array([5, 2, 4, 4]) + array([1, 4, -2, -1])/10
>>> a < b
array([False,  True,  True, False], dtype=bool)
>>> a == c
array([ True,  True,  True,  True], dtype=bool)
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

```
>>> any(a < b)
True
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

```
>>> any(a < b)
True
>>> any(a == b)
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

```
>>> any(a < b)
True
>>> any(a == b)
False
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

```
>>> any(a < b)
True
>>> any(a == b)
False
>>> all(a == c)
```

Para reducir el arreglo de booleanos a un único valor, se puede usar las funciones `any` y `all`.`any` devuelve `True` si al menos uno de los elementos es verdadero, mientras que `all` devuelve `True` sólo si todos lo son (del inglés, `any` signfica *alguno*, y `all` significa *todos*):

```
>>> any(a < b)
True
>>> any(a == b)
False
>>> all(a == c)
True
```

Funciones sobre arreglos

`NumPy` provee muchas funciones matemáticas que también operan elemento a elemento.

Funciones sobre arreglos

NumPy provee muchas funciones matemáticas que también operan elemento a elemento.

```
>>> from numpy import linspace, pi, sin
>>> x = linspace(0, pi/2, 9)
>>> x
array([ 0.        ,  0.19634954,  0.39269908,
        0.58904862,  0.78539816,  0.9817477 ,
        1.17809725,  1.37444679,  1.57079633])
>>> sin(x)
```

Funciones sobre arreglos

NumPy provee muchas funciones matemáticas que también operan elemento a elemento.

```
>>> from numpy import linspace, pi, sin
>>> x = linspace(0, pi/2, 9)
>>> x
array([ 0.        ,  0.19634954,  0.39269908,
        0.58904862,  0.78539816,  0.9817477 ,
        1.17809725,  1.37444679,  1.57079633])
>>> sin(x)
array([ 0.        ,  0.19509032,  0.38268343,
        0.55557023,  0.70710678,  0.83146961,
        0.92387953,  0.98078528,  1.        ])
```

Como puede ver, los valores obtenidos van de 0 a 1, que es justamente como se comporta la función seno en el intervalo $[0, \pi/2]$.

Aquí también se hace evidente otra de las ventajas de los arreglos: al mostrarlos en la consola o al imprimirlos, los valores aparecen perfectamente alineados. Con las listas, esto no ocurre:

```
>>> list(sin(x))
```

Aquí también se hace evidente otra de las ventajas de los arreglos: al mostrarlos en la consola o al imprimirlos, los valores aparecen perfectamente alineados. Con las listas, esto no ocurre:

```
>>> list(sin(x))
[0.0, 0.19509032201612825, 0.38268343236508978, 0.55557
1960218, 0.70710678118654746, 0.83146961230254524, 0.92
3251128674, 0.98078528040323043, 1.0]
```

Arreglos aleatorios

`NumPy` contiene a su vez otros módulos que proveen funcionalidad adicional a los arreglos y funciones básicos.

El módulo `numpy.random` provee funciones para crear números aleatorios (es decir, generados al azar), de las cuales la más usada es la función `random`, que entrega un arreglo de números al azar distribuidos uniformemente entre 0 y 1:

```
>>> from numpy.random import random
```

```
>>> random(3)
```

```
>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
```

```
>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
array([ 0.07405763,  0.04083838,  0.72962968])
```

```
>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
array([ 0.07405763,  0.04083838,  0.72962968])
>>> random(3)
```

```
>>> random(3)
array([ 0.53077263,  0.22039319,  0.81268786])
>>> random(3)
array([ 0.07405763,  0.04083838,  0.72962968])
>>> random(3)
array([ 0.51886706,  0.46220545,  0.95818726])
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
-2.3
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
-2.3
>>> a[-2]
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
-2.3
>>> a[-2]
4.7
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
-2.3
>>> a[-2]
4.7
>>> a[3]
```

Obtener elementos de un arreglo

Cada elemento del arreglo tiene un índice, al igual que en las listas. El primer elemento tiene índice 0. Los elementos también pueden numerarse desde el final hasta el principio usando índices negativos. El último elemento tiene índice -1:

```
>>> a = array([6.2, -2.3, 3.4, 4.7, 9.8])
>>> a[0]
6.2
>>> a[1]
-2.3
>>> a[-2]
4.7
>>> a[3]
4.7
```

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i : j]$. Los índices i y j indican el rango de valores que serán devueltos:

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i : j]$. Los índices i y j indican el rango de valores que serán devueltos:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
```

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i:j]$. Los índices i y j indican el rango de valores que serán devueltos:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
```

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i:j]$. Los índices i y j indican el rango de valores que serán devueltos:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
array([-2.3,  3.4,  4.7])
```

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i : j]$. Los índices i y j indican el rango de valores que serán devueltos:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
array([-2.3,  3.4,  4.7])
>>> a[2:-2]
```

Una sección del arreglo puede ser obtenida usando el operador de rebanado $a[i : j]$. Los índices i y j indican el rango de valores que serán devueltos:

```
>>> a
array([ 6.2, -2.3,  3.4,  4.7,  9.8])
>>> a[1:4]
array([-2.3,  3.4,  4.7])
>>> a[2:-2]
array([ 3.4])
```

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

```
>>> a[:2]
array([ 6.2, -2.3])
```

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

```
>>> a[:2]
array([ 6.2, -2.3])
>>> a[2:]
```

Si el primer índice es omitido, el rebanado comienza desde el principio del arreglo. Si el segundo índice es omitido, el rebanado termina al final del arreglo:

```
>>> a[:2]
array([ 6.2, -2.3])
>>> a[2:]
array([ 3.4,  4.7,  9.8])
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.    ,  0.125,  0.25 ,  0.375,  0.5   ,  0.625,
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.    ,  0.125,  0.25 ,  0.375,  0.5   ,  0.625,
>>> a[1:7:2]
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.    ,  0.125,  0.25 ,  0.375,  0.5   ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.   ,  0.375,  0.75 ])
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.   ,  0.375,  0.75 ])
>>> a[-2::-2]
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.   ,  0.125,  0.25 ,  0.375,  0.5  ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.   ,  0.375,  0.75 ])
>>> a[-2::-2]
array([ 0.875,  0.625,  0.375,  0.125])
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.    ,  0.125,  0.25 ,  0.375,  0.5   ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.    ,  0.375,  0.75 ])
>>> a[-2::-2]
array([ 0.875,  0.625,  0.375,  0.125])
>>> a[::-1]
```

Un tercer índice puede indicar cada cuántos elementos serán incluidos en el resultado:

```
>>> a = linspace(0, 1, 9)
>>> a
array([ 0.    ,  0.125,  0.25 ,  0.375,  0.5   ,  0.625,
>>> a[1:7:2]
array([ 0.125,  0.375,  0.625])
>>> a[::3]
array([ 0.    ,  0.375,  0.75 ])
>>> a[-2::-2]
array([ 0.875,  0.625,  0.375,  0.125])
>>> a[::-1]
array([ 1.    ,  0.875,  0.75 ,  0.625,  0.5   ,  0.375,
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
array([], dtype=float64)
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
array([], dtype=float64)
>>> b[1:5:2]
```

Una manera simple de recordar cómo funciona el rebanado es considerar que los índices no se refieren a los elementos, sino a los espacios entre los elementos:

```
>>> b = array([17.41, 2.19, 10.99, -2.29, 3.86, 11.10])
>>> b[2:5]
array([ 10.99,  -2.29,   3.86])
>>> b[:5]
array([ 17.41,   2.19,  10.99,  -2.29,   3.86])
>>> b[1:1]
array([], dtype=float64)
>>> b[1:5:2]
array([ 2.19, -2.29])
```

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.
>>> a.min()
```

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.
>>> a.min()
-2.5
```

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.
>>> a.min()
-2.5
>>> a.max()
```

Algunos métodos convenientes

Los arreglos proveen algunos métodos útiles que conviene conocer.

Los métodos `min` y `max`, entregan respectivamente el mínimo y el máximo de los elementos del arreglo:

```
>>> a = array([4.1, 2.7, 8.4, pi, -2.5, 3, 5.2
>>> a.min()
-2.5
>>> a.max()
8.4000000000000004
```

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

```
>>> a.argmin()
4
```

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

```
>>> a.argmin()
4
>>> a.argmax()
```

Los métodos `argmin` y `argmax` entregan respectivamente la posición del mínimo y del máximo:

```
>>> a.argmin()
4
>>> a.argmax()
2
```

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

```
>>> a.sum()
24.041592653589795
```

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

```
>>> a.sum()
24.041592653589795
>>> a.prod()
```

Los métodos `sum` y `prod` entregan respectivamente la suma y el producto de los elementos:

```
>>> a.sum()
24.041592653589795
>>> a.prod()
-11393.086289208301
```

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

6. Otras funciones dentro de `numpy.linalg`
   - Función `Eye`
   - Función `reshape`
   - Traza y determinante
   - Inversa de una matriz
   - Matriz transpuesta
   - Valores y vectores propios

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]]))
```