Python 3 para impacientes

"Simple es mejor que complejo" (Tim Peters)

domingo, 7 de julio de 2019

El módulo array frente a las listas Python

El módulo **array** de la librería estándar de Python permite declarar un objeto que es similar a una **lista** pero sólo puede almacenar datos del mismo tipo: números enteros con distintos tamaños y números de punto flotante, entre otros.

Por esta razón cuando se define un **array** lo primero que hay que establecer es el tipo de dato que va a contener con una letra identificativa. Esto que parece una restricción garantiza también, cuando se trabaja con miles o millones de datos, que todos sean del mismo tipo.

Tipos de Arrays

b - char con signo (int, 1 byte)
B - char sin signo (int, 1 byte)
h - short con signo (int, 2 bytes)
H - short sin signo (int, 2 bytes)
i - int con signo (int, 4 bytes)
I - int sin signo (int, 4 bytes)
l - long con signo (int, 8 bytes)
L - long sin signo (int, 8 bytes)
f - float (float, 4 bytes)
d - double (float, 8 bytes)

Las **listas** en cambio son estructuras flexibles que pueden contener datos elementales de distintos tipos y otras estructuras como diccionarios, tuplas o incluso otras listas. Además, como las listas son objetos que forman parte del propio lenguaje, en general, será más rápido añadir y procesar elementos en una lista que en un array. El inconveniente lo vamos a encontrar cuando trabajemos con listas de gran tamaño. Los arrays son mucho más eficientes en este sentido y aprovechan mejor los recursos, en especial, la memoria.

La versatilidad de una lista penaliza a medida que su tamaño va creciendo pero sólo cuando se trata de listas muy grandes. En las pruebas que hemos realizado para poner al límite ambas estructuras hemos observado que agregar elementos a una lista era hasta un 40% más rápido que a un array. Este comportamiento se ha mantenido en el equipo que hemos utilizado para las pruebas hasta alcanzar 650.000.000 elementos, aproximadamente. A partir de ahí, a medida que se iban agotando los recursos, el sistema enlentecía drásticamente y con casi 800.000.000 elementos se interrumpía la ejecución del script produciendo un error de memoria (*MemoryError*) y alguna vez el extraño "*Killed*". En el caso de los arrays como son estructuras más compactas y seguras hemos podido declarar y operar arrays de 5.000.000.000 elementos sin que el sistema se rompiera.

Por lo demás, tanto las listas como los arrays tienen métodos básicos para agregar, insertar, extender, acceder, borrar, contar elementos, etc. La clase **array** incorpora los métodos **fromfile()** y **tofile()** que permiten leer de un fichero o escribir a un fichero los elementos de un array; aunque para operaciones más complejas y arrays con más de una dimensión lo recomendable es la biblioteca **Numpy**.

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...

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...

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...

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...

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

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...

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...

Threading: programación con hilos (I)
En programación, la técnica que permite que una aplicación ejecute

A continuación mostramos el uso del módulo **array** con algunos ejemplos que servirán para constatar el parecido con las listas, exceptuando el modo en que se declaran y que hay algunos métodos diferentes.

Declarar arrays

```
# Importar módulo array
import array

# Declarar lista con valores numéricos
lista1 = [1, 0, 1, 0, 1, 1, 0, 0]

# Obtener cadena con todos los tipos de arrays disponbiles
array.typecodes  # 'bBuhHiIlLqQfd'

# Declarar 'array1' de tipo 'char sin signo' con datos de 'lista1'
array1 = array.array('B', lista1)

# Declarar 'array2' de tipo 'float' con datos de 'lista1'
array2 = array.array('f', lista1)

# Declarar 'array3' de tipo 'int sin signo' con datos de
# 'lista1' en orden inverso
array3 = array.array('I', (elemento for elemento in lista1[::-1]))

# Declarar 'array4' de tipo 'char con signo' a partir cadena de bytes
cadena = b'Python'
array4 = array.array('b', cadena)

# Declarar 'array5' de tipo 'int con signo' con valores del 0 al 9
array5 = array.array('i', range(10))

# Obtener tipo de array y datos de 'array1'
array1  # array('B', [1, 0, 1, 0, 1, 1, 0, 0])

# Obtener tipo de array y datos de 'array2'
array2  # array('f', [1.0, 0.0, 1.0, 0.0, 1.0, 1.0, 0.0, 0.0])

# Obtener tipo de array y datos de 'array3'
array3  # array('I', [0, 0, 1, 1, 0, 1, 0, 1])

# Obtener tipo de array y datos de 'array4'
array4  # array('b', [80, 121, 116, 104, 111, 110])

# Obtener tipo de array y datos de 'array4'
array5  # array('i', [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

# Obtener tipo de objeto de 'array1'
type(array1)  # array.array

# Obtener tipo de array de 'array1', 'array2' y 'array2'
array1.typecode  # 'B'
array2.typecode  # 'f'
array3.typecode  # 'I'

# Obtener tamaño (en bytes) que ocupa un elemento en los arrays
array1.itemsize  # 1
array2.itemsize  # 4
array3.itemsize  # 4
```

Operaciones básicas con arrays

```
# Añadir nuevos elementos a 'array1' y 'array2'
array1.append(1)
array1.append(0)
array2.append(1)

# Obtener número de elementos de 'array1', 'array2' y 'array3'
len(array1)  # 10
len(array2)  # 9
len(array3)  # 8

# Obtener tupla con dirección de memoria actual de los arrays y
# número de elementos (Útil sólo para operaciones de bajo nivel).
array1.buffer_info()  # (140391309074144, 10)
```

simultáneamente varias operaciones en el mismo espacio de proceso se...

Archivo

julio 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

```
array2.buffer_info()  # (140391299217200, 9)
array3.buffer_info()  # (140391307750896, 8)

# Obtener el número de veces que se repite un valor en un array
array1.count(1)  # 5
array2.count(1)  # 5
array3.count(0)  # 4

# Obtener posición donde aparece por primera vez elemento buscado
array1.index(1)  # 0
array2.index(1)  # 0
array3.index(1)  # 2

# Insertar un elemento delante de la posición indicada en 'array1'
array1.insert(0, 10)
array1.insert(2, 20)
array1  # array('B', [10, 1, 20, 0, 1, 0, 1, 1, 0, 0, 1, 0])

# Borrar el último y tercer elemento en 'array1'
array1.pop()
array1.pop(2)
array1  # array('B', [10, 1, 0, 1, 0, 1, 1, 0, 0, 1])

# Borrar el primer elemento en 'array1' que coincida con el indicado

array1.remove(10)
array1  # array('B', [1, 0, 1, 0, 1, 1, 0, 0, 1])

# Invertir el orden de los elementos en 'array1'
array1.reverse()
array1  # array('B', [1, 0, 0, 1, 1, 0, 1, 0, 1])

# Extender o añadir elementos a un array
array3  # array('I', [0, 0, 1, 1, 0, 1, 0, 1])
array3.extend([0, 0])
array3  # array('I', [0, 0, 1, 1, 0, 1, 0, 1, 0, 0])

# Obtener el primer, tercer y último elemento de 'array3'
array3[0]  # 0
array3[2]  # 1
array3[-1]  # 0

# Cambiar el valor a 1 del primer y penúltimo elemento de 'array3'
array3[0] = 1
array3[-2] = 1
```

Cadenas de bytes. Lectura/escritura de arrays de/a archivos

```
# Asignar con frombytes() una cadena de bytes a 'array1'
array1 = array.array('b')
cadena = b'Arrays Python'
array1.frombytes(cadena)
array1  #  array('b', [65, 114, 114, 97, 121, 115, 32, 80, 121,
        #              116, 104, 111, 110])

# Escribir el contenido de 'array1' a un archivo
cadena = b'Arrays Python'
array1 = array.array('b', cadena)
archivo = open('datos.txt', 'wb')
array1.tofile(archivo)
archivo.close

# Leer y añadir a 'array1' los 6 primeros bytes del archivo anterior
array1 = array.array('b')
archivo = open('datos.txt', 'rb')
array1.fromfile(archivo, 6)
array1  # array('b', [65, 114, 114, 97, 121, 115])

# Convertir a bytes los valores leídos con anterioridad
cadena = array1.tobytes()
cadena  # b'Arrays'

# Convertir bytes a cadena de texto
cadena = cadena.decode()
cadena  # 'Arrays'

# Intentar añadir elementos de una lista a un array que
# contiene elementos. Cuando se produce un error de tipo
```

```
# el array mantiene sus datos originales.
lista1 = [0, 1, 3.31, 3.5, 1.9, 0, False]
array1 = array.array('b', [9, 10, 11])
try:
    array1.fromlist(lista1)
except:
    print('Error al intentar agregar lista1')
finally:
    print(array1)
```

Relacionado:

- Cadenas, listas, tuplas, diccionarios y conjuntos

Ir al índice del tutorial de Python

Publicado por Pherkad en 11:19

Etiquetas: Python3

Entrada más reciente Inicio Entrada antigua