Python

Lic. Mariela Asensio

Estructuras de datos

Hasta ahora hemos trabajado con tipos de datos simples, es decir aquellos que guardan un único valor determinado.

Python ofrece diversos tipos de datos complejos. Los tipos de datos complejos almacenan más de un elemento, por esto se denominan también “contenedores”.

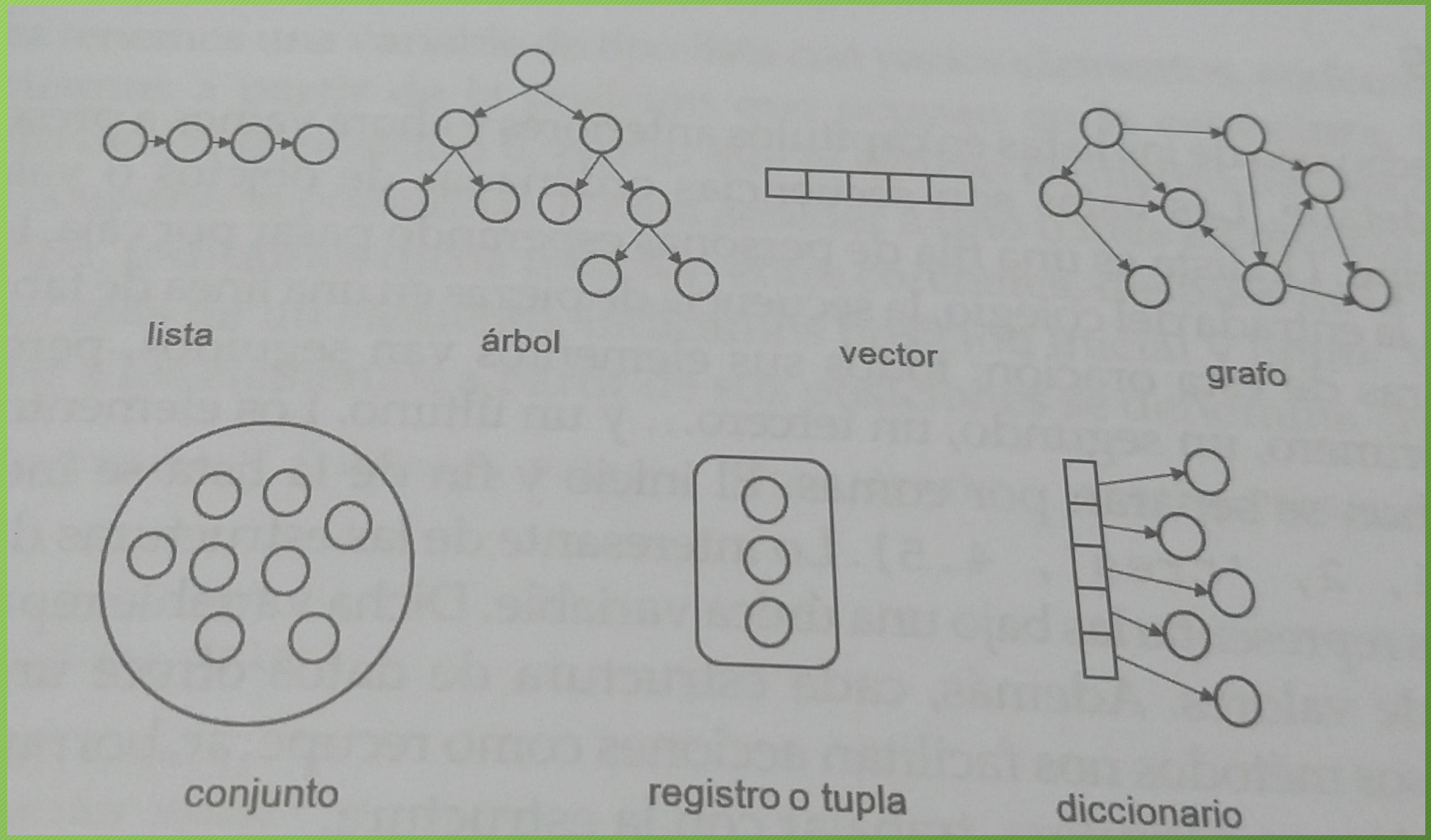
Estructura de Datos

Una estructura de datos es una forma de organizar, gestionar y almacenar conjuntos de datos para acceder a ellos y manipularlos de manera eficiente de acuerdo con el problema que estamos resolviendo.

Python ofrece de forma nativa – sin necesidad de invocar bibliotecas – cuatro estructuras de datos:

- Listas
- Tuplas
- Conjuntos
- Diccionarios

Las estructuras de datos nos facilitan su agrupamiento y las estructuras de control como while o for, su procesamiento.



Listas

Una lista es una colección de datos que pueden tener diferente tipo. Los datos se escriben entre corchetes separados por comas:
[dato1, dato2, dato3, …, dato4]

Una lista es una estructura de datos dinámica cuyos componentes se pueden modificar.
Las celdas son enumeradas iniciando en cero.
Se puede acceder a los componentes de una lista mediante un índice entre corchetes.
En la notación de índices se puede especificar el acceso a varios elementos o componentes mediante un rango para el índice. El rango no incluye el extremo derecho especificado.

La representación visual de la lista se puede describir con el siguiente gráfico, en el que los componentes se almacenan en celdas enumeradas desde cero.

‘abc’	73	5.28	‘rs’	50
0	1	2	3	4

```
>>> x = ['abc', 73, 5.28, 'rs', 50]
```

```
>>> x[0]
'abc'
```

Muestra el primer componente de la lista

```
>>> x[2]
5.28
```

Muestra el elemento que ocupa la posición 2

```
>>> x[1:4]
[73, 5.28, 'rs']
```

Muestra un rango que va de la posición 1 a la 4, pero no muestra el último elemento.

```
x[6]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    x[6]
IndexError: list index out of range
```

Un índice fuera de rango genera un error.

```
>>> x[2: ]
[5.28, 'rs', 50]
```

Muestra los componentes desde la posición 2 hasta el final.

```
>>> x[ : 3]
['abc', 73, 5.28]
```

Muestra los componentes desde el inicio hasta la posición 2.

```
>>> x[0:5:2]
['abc', 5.28, 50]
```

Muestra todos los componentes que ocupan posiciones pares. El tercer elemento es el incremento.

```
>>> x[-1]
50
```

Muestra el último elemento.

```
>>> x[-2]
'rs'
```

Muestra el penúltimo elemento.

```
>>> x[ : -2]
['abc', 73, 5.28]
```

Muestra los elementos desde el inicio menos los últimos 2 .

```
>>> x[-2: ]
['rs', 50]
```

Muestra los elementos desde penúltimo hasta el final.

```
>>> x[1]=45
>>> x
['abc', 45, 5.28, 'rs', 50]
```

La lista fue modificada.

```
>>> x=[20,30,40,50,60,70,80,90]
>>> x[::2]
[20, 40, 60, 80]
```

El tercer elemento indica el incremento

```
>>> x[::3]
[20, 50, 80]
```

```
>>> x[::-1]
[90, 80, 70, 60, 50, 40, 30, 20]
```

Invierte la lista.

Listas anidadas

Una lista puede contener elementos tipo lista

```
>>> x =[123, 'Mariela', [5,10,15], 5, 80.2]
>>> x[0]
123
>>> x[1]
'Mariela'
>>> x[2]
[5, 10, 15]
```

```
>>> x[2][2]
15
```

Se requiere un segundo índice para referirse a los elementos del componente tipo lista

Se pueden modificar los elementos de la lista.

```
x[2][1]=8
>>> x
[123, 'Mariela', [5, 8, 15], 5, 80.2]
```

Los elementos de una lista se pueden desempacar asignando variables.

```
>>> x = [123, 'Mariela', [5, 8, 15], 5, 80.2]
>>> a,b,c,d,e= x
>>> a
123
>>> b
'Mariela'
>>> c
[5, 8, 15]
>>> d
5
>>> e
80.2
```

Lista creada a partir de una cadena de caracteres

```
>>> x = list('Murcielago')
>>> x
['M', 'u', 'r', 'c', 'i', 'e', 'l', 'a', 'g', 'o’]
>>> x[5:]
['e', 'l', 'a', 'g', 'o']
>>> x[:5]
['M', 'u', 'r', 'c', 'i']
```

Eliminar elementos

La forma más directa para eliminar un elemento dentro de la lista es mediante la instrucción del

```
>>> l = list(range(2010,2018))
>>> l
[2010, 2011, 2012, 2013, 2014, 2015, 2016, 2017]
>>> del l[3]
>>> l
[2010, 2011, 2012, 2014, 2015, 2016, 2017]
```

Funciones para el manejo de listas numéricas

```
>>> l = list(range(10,100,10))
>>> l
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
>>> len(l)
9
```

Longitud de una lista

```
>>> max(l)
90
```

Mayor valor de una lista

```
>>> min(l)
10
```

Menor valor de una lista

```
>>> sum(l)
450
```

Suma de los elementos de una lista

Funciones para el manejo de listas

List.append(x) Añade el elemento x al final de la lista

```
>>> l.append(150)
>>> l
[10, 20, 30, 40, 50, 60, 70, 80, 90, 150]
```

List.extend(iterable) Añade al final de la lista todo los elementos del iterable indicado como argumento.

```
>>> l.extend(range(4))
>>> l
[10, 20, 30, 40, 50, 60, 70, 80, 90, 150, 0, 1, 2, 3]
```

List.insert(i,x) Inserta un nuevo elemento x en la posición i de la lista

```
>>> l.insert(5,55)
>>> l
[10, 20, 30, 40, 50, 55, 60, 70, 80, 90, 150, 0, 1, 2, 3]
```

List.remove(x) Elimina de la lista el primer – y sólo el primer – elemento con el valor x

```
>>> l.remove(150)
>>> l
[10, 20, 30, 40, 50, 55, 60, 70, 80, 90, 0, 1, 2, 3]
```

List.pop([i]) Saca el elemento que ocupa la posición i en la lista y devuelve su valor. Si se utiliza esta función sin argumentos saca el último elemento de la lista.

```
>>> l.pop(3)
40
>>> l
[10, 20, 30, 50, 55, 60, 70, 80, 90, 0, 1, 2, 3]
```

List.clear() Vacía la lista, eliminando todos sus elementos

```
>>> l.clear()
>>> l
[]
```

List.index(x) Devuelve la posición que ocupa el valor x en la lista. Devuelve la posición de la primera ocurrencia.

```
>>> l=list(range(4,14,2))
>>> l
[4, 6, 8, 10, 12]
>>> l.index(8)
2
```

List.count(x) Cuenta el número de elementos de la lista con el valor x

```
>>> l.count(6)
1
>>> l.append(6)
>>> l
[4, 6, 8, 10, 12, 6]
>>> l.count(6)
2
```

List.sort(key=None, reverse=False) Modifica la lista ordenando sus elementos. Los elementos Key y Reverse son opcionales. Sus valores por defecto son None y False.

```
>>> l=[9,1,8,2,4,3,7,6,5]
>>> l.sort()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> l.sort(reverse=True)
>>> l
[9, 8, 7, 6, 5, 4, 3, 2, 1]
```

List.reverse() Invierte el contenido de una lista

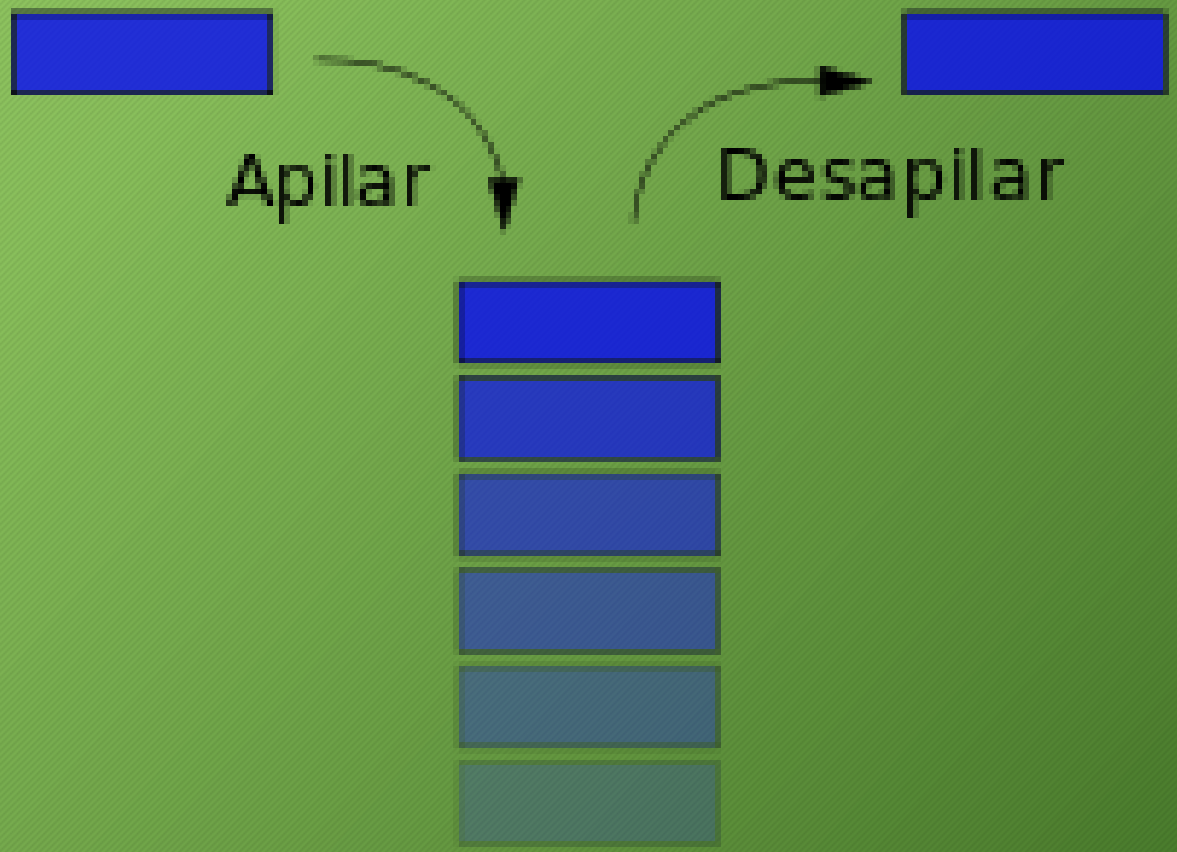
```
>>> l.reverse()
>>> l
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List.copy() Copia el contenido de una lista

```
>>> l1 = l.copy()
>>> l1
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Pilas

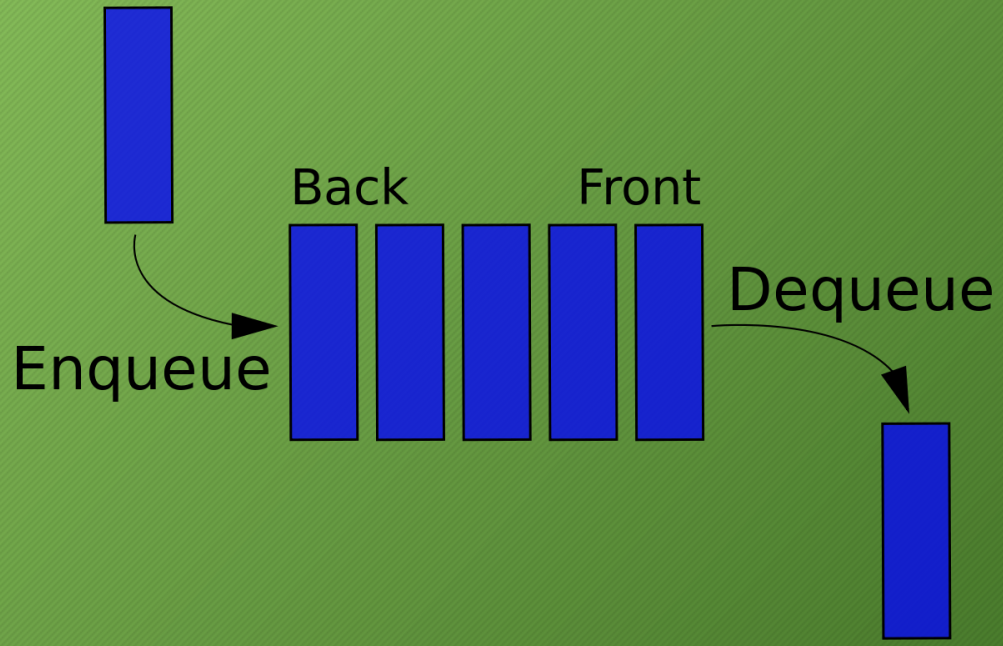
Una estructura de datos tipo pila, es una lista en la que los elementos se añaden al final y se sacan del final. Se extrae siempre el elemento añadido más recientemente. LIFO
Para que una lista funcione como una pila basta con usar append() al añadir elementos y pop() para sacarlos.



Colas

También es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado. FIFO: First In First Out. Para que una lista funcione como cola podemos usar insert(0,X) para añadir elementos y pop() para sacar los más antiguos. Aunque estas opciones no son eficientes.

Para implementar una cola, podemos usar **`collections.deque`** el cual fue diseñado para agregar y sacar de ambas puntas de forma rápida.



```
>>> from collections import deque
>>> cola = deque(['Mariela','Luciana','Maria Marta','Lucas'])
>>> cola.append('Valeria')
>>> cola
deque(['Mariela', 'Luciana', 'Maria Marta', 'Lucas', 'Valeria'])
>>> cola.append('Diego')
>>> cola
deque(['Mariela', 'Luciana', 'Maria Marta', 'Lucas', 'Valeria', 'Diego'])
>>> cola.popleft()
'Mariela'
>>> cola
deque(['Luciana', 'Maria Marta', 'Lucas', 'Valeria', 'Diego'])
>>> cola.popleft()
'Luciana'
>>> cola
deque(['Maria Marta', 'Lucas', 'Valeria', 'Diego'])
```

Compresión de Listas

Iterar sobre los elementos de una lista, procesarlos y generar nuevas listas, es algo que implementaremos de varias maneras y en repetidas ocasiones.
Para simplificar algunas operaciones sobre listas, Python ofrece una sintaxis más compacta, para generar listas a partir de otras mediante iteración. Esta forma de trabajar con listas se denomina “compresión de listas”.

```
>>> cuadrados = []
>>> for x in range(10):
    cuadrados.append(x**2)

>>> cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Podemos calcular la lista de cuadrados sin ningún efecto secundario haciendo:

```
>>> cuadrados = [x ** 2 for x in range(10)]
>>> cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
combs = []
for x in [1,2,3]:
    for y in [3,1,4]:
        if x != y:
            combs.append((x, y))
print(combs)
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Otro ejemplo

```
edades = [23, 32, 19, 22, 25, 30, 27]
mayores=[x for x in edades if x > 25]
print('hay', len(mayores),'mayores de 25 años')
print(mayores)

hay 3 mayores de 25 años
[32, 30, 27]
```

Otro ejemplo anidando bucles

```
chicos = ['Juan', 'Felipe', 'Fausto']
chicas = ['Marta', 'Cecilia', 'Luciana']
parejas_posibles = [(x,y) for x in chicas for y in chicos]
print(parejas_posibles)
[('Marta', 'Juan'), ('Marta', 'Felipe'), ('Marta', 'Fausto’),
('Cecilia', 'Juan'), ('Cecilia', 'Felipe'), ('Cecilia', 'Fausto’),
('Luciana', 'Juan'), ('Luciana', 'Felipe'), ('Luciana', 'Fausto')]
```

Enumeración de listas

Entrega una lista indexada

```
>>> dias = ['lunes', 'martes', 'miércoles', 'jueves', 'viernes']
>>> list(enumerate(dias))
[(0, 'lunes'), (1, 'martes'), (2, 'miércoles'), (3, 'jueves'), (4, 'viernes')]
```

```
>>> for i,e in enumerate(dias):
    print(i,e)

0 lunes
1 martes
2 miércoles
3 jueves
4 viernes
```

Arreglos unidimensionales - vectores

Los arreglos son listas de una o más dimensiones cuyos elementos son del mismo tipo, usualmente numérico. Su manejo se realiza principalmente con los operadores y funciones de la librería **NumPy**.

NumPy es una librería de soporte para aplicaciones en álgebra lineal, estadística y otras áreas de las matemáticas e ingeniería.

Un vector es un arreglo de una dimensión, es decir, una lista con datos del mismo tipo.

El manejo de los índices admiten formas especiales y su utilización se denomina "**slicing**".

Para definir arreglos se usa la palabra especial **array()**

```
from numpy import*
x = array([10, 20, 30, 40, 50])
print(x)
```

Los arreglos se almacenan en celdas numeradas desde cero y sus componentes se manejan mediante índices como las listas comunes.

10	20	30	40	50
0	1	2	3	4

Creación de arreglos con especificación de tipo

```
import numpy as np
x = np.array([10, 20, 30, 40, 50],dtype=int)
print(x)
```

```
[10 20 30 40 50]
```

Si no se indica el tipo lo toma del contenido, si hay un componente real, tomará todo como real

Operaciones con arreglos

El operador + suma el valor a cada elemento

```
import numpy as np
a = np.array([10, 20, 30, 40, 50], int)
a = a + 4
print(a)

[14 24 34 44 54]
```

Suma en forma correspondiente

```
import numpy as np
a = np.array([10, 20, 30, 40, 50], int)
a = a + [4,6,8,10,12]
print(a)
[14 26 38 50 62]
```

El operador * se aplica a cada elemento

```
import numpy as np
a = np.array([10, 20, 30, 40, 50], int)
a = 2*a
print(a)

[ 20  40  60  80 100]
```

Funciones para llenar un arreglo

```
import numpy as np
a = np.array([1,2,3,4], int)
a.fill(5)
print(a)

[5 5 5 5]
```

Llena el arreglo con 5's

```
import numpy as np
a = np.zeros(5,int)
print(a)

[0 0 0 0 0]
```

Llena con ceros enteros

```
import numpy as np
a = np.zeros(5,float)
print(a)

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

Llena con ceros reales

```
import numpy as np
a = np.ones(5,int)
print(a)

[1 1 1 1 1]
```

Llena con unos

Función empty

La función empty entrega un arreglo con las dimensiones y tipo indicados, pero sin especificar el contenido.

```
import numpy as np
a = np.empty(5,int)
print(a)

[0 0 0 0 0]
```

```
import numpy as np
a = np.empty(5,str)
print(a)

['' '' '' '' '']
```

Definición de arreglos mediante declaraciones implícitas

```
import numpy as np
a = np.array(range(5))
print(a)

[0 1 2 3 4]
```

```
import numpy as np
from random import*
a = np.array([randint(0,9)for i in range(5)])
print(a)
[0 3 6 8 7]
```

Construcción de un arreglo con números aleatorios

Funciones de NumPy para declarar rangos de arreglos

```
import numpy as np
a = np.arange(5)
print(a)
[0 1 2 3 4]
```

La función rango de Numpy

```
import numpy as np
a = np.arange(1,10,2)
print(a)
[1 3 5 7 9]
```

Se puede indicar el inicio, final e incremento

```
import numpy as np
a = np.arange(1,2,0.2)
print(a)
[1.  1.2 1.4 1.6 1.8]
```

Admite valores con decimales

```
import numpy as np
a = np.arange(1,2,0.2)
2*a+1
print(a)
[1.  1.2 1.4 1.6 1.8]
```

Se puede operar con el arreglo

Linspace()

```
import numpy as np
a = np.linspace(10,20,5)
print(a)
[10.  12.5 15.  17.5 20. ]
```

Genera 5 números equiespaciados entre 10 y 20

```
import numpy as np
a = np.linspace(0.1,0.2,6)
print(a)
[0.1  0.12 0.14 0.16 0.18 0.2 ]
```

Clip()

Una función para poner límites a los valores de una arreglo. Los valores fuera del rango son recortados.

```
import numpy as np
a = np.array([2,9,30,10,5,7,8,9,3,20,8,4])
b=np.clip(a,4,9)
print(b)
```

```
[4 9 9 9 5 7 8 9 4 9 8 4]
```

Funciones matemáticas de NumPy para arreglos

Función	Aplicación
np.prod(x)	Producto de los componentes
np.argmax(x)	Índice del mayor valor
np.argmin(x)	Índice del menor valor
np.sum(x)	Suma de componentes
np.consum(x)	Suma acumulada (el resultado es un arreglo)
np.sort(x)	Ordenamiento de un arreglo
np.diff(x)	Restas sucesivas

```
import numpy as np
a = np.array([3,4,9,7,2,9,6])
b=np.prod(a)
print('Producto de los componentes: ',b)
c=np.argmin(a)
print('Indice del menor valor: ',c)
d=np.argmax(a)
print('Indice del mayor valor: ',d)
e=np.sum(a)
print('La suma de los elementos: ',e)
f=np.cumsum(a)
print('La suma acumulada: ',f)
g=np.sort(a)
print('Ordenado: ‘,g)
```

```
Producto de los componentes:  81648
Indice del menor valor:  4
Indice del mayor valor:  2
La suma de los elementos:  40
La suma acumulada:  [ 3  7 16 23 25 34 40]
Ordenado:  [2 3 4 6 7 9 9]
```

Formas especiales de selección de elementos de los arreglos

Los arreglos admiten formas especiales para seleccionar los componentes. Estas formas especiales no están disponible para el manejo de listas.

```
import numpy as np
a = np.array([10,14,25,50])
e=a>20
print(e)

[False False  True  True]
```

Expresión lógica para seleccionarlos elementos con valor mayor a 20. El resultado son valores lógicos

```
import numpy as np
a = np.array([10,14,25,50])
e=a>20
print(e)
c=np.sum(e)
print(c)
print(a[e])
a[a>20]=100
print(a)

[False False  True  True]
2
[25 50]
[ 10  14 100 100]
```

Los valores lógicos se pueden sumar

Los valores lógicos actúan como índices

Se pueden modificar los valores de los elementos seleccionados

Uso de la función **extract** para seleccionar elementos de un arreglo

```
import numpy as np
a = np.array([10,14,25,50])
u=np.extract(a>20,a)
print(u)

[25 50]

import numpy as np
a = np.array([10,14,25,50])
u=np.sum(np.extract(a>20,a))
print(u)

75
```

Se pueden usar índices para seleccionar elementos del arreglo

```
import numpy as np
a = np.array([10,14,25,50])
i = [0,2,1,1]
print(a[i])
```

```
[10 25 14 14]
```

Un arreglo puede usarse para seleccionar elementos de otro arreglo

```
import numpy as np
a = np.array([3,5,7,4,8])
b = np.array([2,5,4,7,9])
r =np.sum(a[b>4])
print(r)
c=list(a[b==a])
print(c)
```

```
17
[5]
```

Funciones para seleccionar elementos con los cuantificadores lógicos

```
import numpy as np
a = np.array([4,3,6,7,3])
r = np.any(a>5)
print(r)
re = np.all(a>5)
print(re)

True
False
```

El resultado es verdadero si al menos uno de los elementos cumple la condición.

El resultado es verdadero si todos cumplen la condición.

Función where

```
import numpy as np
a = np.array([23,45,38,27,62])
r = np.where(a>30)
print(r)
```

```
(array([1, 2, 4], dtype=int64),)
```

El resultado es un arreglo de índices

```
x=a[r]
print(x)
```

```
[45 38 62]
```

Muestra los elementos seleccionados

La función where es muy útil para sustituir valores en un arreglo

Sintaxis: where(condición,x,y)

Los valores que cumplen la condición son sustituidos por el valor x.
Los otros son sustituidos por el valor de y

```
import numpy as np
a = np.array([3,5,7,4,8])
e = np.where(a>4,0,9)
print(e)
```

```
[9 0 0 9 0]
```

```
import numpy as np
a = np.array([3,5,7,4,8])
e = np.where(a%2==0,'si','no')
print(e)
```

```
['no' 'no' 'no' 'si' 'si']
```

Cada valor de a que es par se sustituye por ‘si’, el resto por ‘no’

Funciones NumPy para modificar arreglos

```
import numpy as np
a = np.array([5,3,7,8])
a = np.concatenate([a,[6]])
print(a)
```

```
[5 3 7 8 6]
```

Concatena arreglos

```
import numpy as np
a = np.array([5,3,7,8])
a=np.delete(a,2)
print(a)
```

```
[5 3 8 6]
```

Elimina elementos de un arreglo de acuerdo a una posición dada.

```
import numpy as np
a = np.array([5,3,7,8])
a=np.insert(a,1,[7])
print(a)
```

```
[5 7 3 7 8]
```

Insertar arreglos en una posición dada.

```
import numpy as np
a = np.array([10,20,30])
b = np.array([50,70])
c = np.concatenate([a,b])
print(c)
```

```
[10 20 30 50 70]
```

Concatena arreglos

```
t = np.array([40,80])
c = np.append(c,t)
print(c)
```

```
[10 20 30 50 70 40 80]
```

Agrega un arreglo al final del otro

La función roll permite girar un arreglo sobre sí mismo

```
import numpy as np
a = np.array([10,20,30,40,50,60,70])
c = np.roll(a,3)
print(c)
```

```
[50 60 70 10 20 30 40]
```

Puede elegirse la cantidad de posiciones a rotar