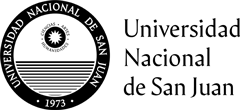
****



Alumno: Enzo Kokot

Dni: 39008472

Carrera: LCC/LSI/TUPW

Registro: E010-194/ E009-102/ E014-129

Alumno: JORGE LLORET

Dni: 33185501

Carrera: LCC

Registro: E010-89

**Facultad de Ciencias Exactas Físicas y Naturales**

Trabajo de investigación:

*Lista Python y Arreglos numpy*

Cátedra:

Programación Orientada a Objetos

Año 2025

Trabajo de investigación lista de python y arreglo de numpy

**Introducción**

En el presente trabajo de investigación se analizan y describen en detalle dos tipos de datos estructurados utilizados en programación: las listas de Python y los arreglos de NumPy. Se abordarán sus características principales, diferencias, ventajas y posibles aplicaciones dentro del desarrollo de software, con el objetivo de comprender su funcionalidad y utilidad.

**Palabras claves:**

Numpy - np.array - lista - python - arreglos

**Definición de Arreglo de Numpy**

NumPy es el paquete fundamental para la computación científica en Python. Es una biblioteca que proporciona un objeto de arreglo multidimensional.

Además, varios objetos derivados (como arreglos enmascarados y matrices), y una variedad de rutinas para operaciones rápidas en arreglos, incluyendo matemáticas, lógica, manipulación de forma, ordenamiento, selección, E/S, transformadas de Fourier discretas, álgebra lineal básica, operaciones estadísticas básicas, simulación aleatoria y mucho más.

**Definición de lista de python.**

Las listas en Python son estructuras de datos heterogéneas y dinámicas que forman parte del núcleo del lenguaje. Las listas son secuencias mutables que pueden contener elementos de diferentes tipos y cuya implementación interna utiliza arrays dinámicos.

|  |  |  |
| --- | --- | --- |
|  | **Listas de Python** | **Arreglos de NumPy** |
| **Tipo de datos** | Puede mezclar tipos (heterogénea) | Homogéneo por diseño, pero puede contener objetos si se define dtype=object |
| **Flexibilidad** | Muy flexible (agregar, borrar, mezclar) | Menos flexible con objetos personalizados |
| **Acceso a elementos** | Acceso por índice (lista[i]) | Acceso por índice (arreglo[i]) |
| **Inserción y eliminación** | Fácil: append(), insert(), remove() | Poco práctico con objetos (requiere copiar el array) |
| **Uso principal** | Datos variados | Vectorizado (rápido, pero sólo para tipos numéricos) |
| **Rendimiento con grandes volúmenes** | Menor rendimiento | Alto rendimiento para datos numéricos |
| **Soporte para objetos personalizados** | Totalmente compatible | Compatible si se usa dtype=object, pero pierde ventajas de NumPy |
| **Tamaño** | Dinámico (crece o decrece fácilmente) | Tamaño fijo, requiere crear nuevo arreglo para cambiar tamaño |
| **Simplicidad y legibilidad** | Más simple para programadores nuevos | Más técnico, requiere saber NumPy |
| **Integración con librerías científicas** | No es directa | Alta (scikit-learn, pandas, etc.) |

Análisis del Código Base

El código presentado implementa un sistema de gestión de productos que utiliza un arreglo NumPy para almacenar instancias de la clase Producto. Se identificaron varios aspectos relevantes:

1. El arreglo se declara con dtype=Producto, lo que técnicamente se traduce a dtype=object, eliminando las ventajas de homogeneidad de NumPy.
2. La operación self.\_\_datos+indiceInflacion no aplica el método \_\_add\_\_ de cada producto como se esperaría, demostrando una limitación en la integración objeto-arreglo.
3. Sobrecarga de memoria: El uso de arreglos NumPy con objetos Python incurre en una sobrecarga similar a las listas pero sin sus ventajas operacionales.

*# se declara la clase Producto*

*class Producto:*

*\_\_descripcion: str*

*\_\_cantidad: int*

*\_\_precioUnitario: float*

*def \_\_init\_\_(self, descripcion, cantidad, precioUnitario):*

*self.\_\_descripcion=descripción*

*self.\_\_cantidad=cantidad*

*self.\_\_precioUnitario=precioUnitario*

*def \_\_add\_\_(self, incremento):*

*self.\_\_precioUnitario=self.\_\_precioUnitario\*(1+incremento/100)*

*def \_\_str\_\_(self):*

*return f"descripción: {self.\_\_descripcion}, precio: {self.\_\_precioUnitario}"*

***# Utilizando Numpy***

*from claseProducto import Producto*

*# se importa la librería de numpy para trabajar con ella*

*import numpy as np*

*class ManejadorArreglo:*

*\_\_datos: None*

*\_\_cantidad: int*

*def \_\_init\_\_(self, cantidad):*

*# Se crea un* ***arreglo de NumPy vacío*** *donde se van a guardar objetos del tipo Producto*

*self.\_\_datos=np.empty(cantidad, dtype=Producto)*

*self.\_\_cantidad=cantidad*

*def test(self):*

*unProducto=Producto("Azúcar", 400, 867.50)*

*self.\_\_datos[0]=unProducto*

*otroProducto=Producto("Harina", 100, 1250)*

*self.\_\_datos[1]=otroProducto*

*ultimoProducto=Producto("Arroz", 120, 2130)*

*self.\_\_datos[2]=ultimoProducto*

*def incremento(self,indiceInflacion):*

*self.\_\_datos+indiceInflacion*

*def mostrarElementos(self):*

*#Muestra los elementos*

*for indice in range(self.\_\_cantidad):*

*print(self.\_\_datos[indice])*

***# Utilizando Lista***

*from claseProducto import Producto*

*class ManejadorLista:*

*def \_\_init\_\_(self):*

*# Crea un atributo privado llamado \_\_datos, y lo inicializa como una* ***lista vacía****.*

*self.\_\_datos = []*

*def test(self):*

*# Se usa el método append() para* ***agregar un nuevo elemento al final de la lista*** *\_\_datos.*

*self.\_\_datos.append(Producto("Azúcar", 400, 867.50))*

*self.\_\_datos.append(Producto("Harina", 100, 1250))*

*self.\_\_datos.append(Producto("Arroz", 120, 2130))*

*def incremento(self, indiceInflacion):*

*#Este método aplica un incremento a todos los productos*

*for i in range(len(self.\_\_datos)):*

*self.\_\_datos[i] + indiceInflacion*

*def mostrarElementos(self):*

*for prod in self.\_\_datos:*

*print(prod)*

*from claseProducto import Producto*

*from arregloPoductos import ManejadorArreglo*

*def test():*

*unManejador=ManejadorArreglo(3)*

*unManejador.test()*

*unManejador.incremento(2.4)*

*unManejador.mostrarElementos()*

*if \_\_name\_\_=='\_\_main\_\_':*

*test()*

Conclusiones

Basado en la evidencia recolectada de múltiples fuentes académicas y técnicas, se concluye que:

1. Para el manejo de objetos de clases personalizadas como en el ejemplo de Producto, las listas nativas de Python son preferibles por su mayor compatibilidad.
2. Los arreglos NumPy no proporcionan beneficios significativos cuando se trabaja con objetos personalizados.
3. El código analizado podría mejorarse sustituyendo el arreglo NumPy por una lista Python, obteniendo mayor claridad y funcionalidad sin pérdida de rendimiento. Esto se propone en el análisis del código base.