Taller de Fundamentos de Programación

Arreglos y Funciones en Python

1. Arreglos (Listas en Python)

1.1 ¿Qué son los arreglos?

Los arreglos, conocidos en Python como **listas**, son una de las estructuras de datos fundamentales en programación. Imagina una caja con múltiples compartimentos numerados donde puedes guardar diferentes objetos. Así funcionan los arreglos: te permiten almacenar múltiples valores en una sola variable.

En el mundo real, usamos arreglos constantemente sin darnos cuenta. Cuando haces una lista de compras, cuando escribes los ingredientes de una receta, o cuando registras las calificaciones de varios exámenes, estás creando arreglos.

¿Por qué son importantes?

- **Organización**: Permiten agrupar datos relacionados bajo un solo nombre
- **Eficiencia**: Facilitan el procesamiento de múltiples valores con pocas líneas de código
- **Acceso aleatorio**: Podemos acceder directamente a cualquier elemento sabiendo su posición

Características principales:

- **Indexados**: Cada elemento tiene una posición numerada (índice) que comienza desde 0. El primer elemento está en la posición 0, el segundo en la 1, y así sucesivamente.
- **Mutables**: Podemos modificar sus elementos después de crearlos, a diferencia de los strings donde no podemos cambiar caracteres individuales.
- **Heterogéneos**: Pueden contener diferentes tipos de datos en una misma lista (números, textos, booleanos, incluso otras listas).
- **Dinámicos**: Pueden cambiar de tamaño durante la ejecución del programa, añadiendo o eliminando elementos.

Visualización de un arreglo:

```
Índices:  0    1    2    3    4
Valores: [10,  20,  30,  40,  50]
```

1.2 Sintaxis básica

Veamos ahora cómo trabajar con listas en Python:

```python
# Creación de una lista vacía
mi_lista = []

# Lista con elementos
numeros = [1, 2, 3, 4, 5]  # Lista de números
nombres = ["Juan", "Pedro", "María"]  # Lista de strings
mixta = [1, "Hola", True, 3.14]  # Lista con diferentes tipos de datos
```

Accediendo a los elementos

Para acceder a un elemento específico de la lista, usamos su índice entre corchetes:

```
# Acceder a elementos por su índice
primer_elemento = numeros[0]  # Obtiene el 1
ultimo_elemento = nombres[2]  # Obtiene "María"

# Los índices negativos cuentan desde el final
ultimo = numeros[-1]  # Obtiene el 5 (último elemento)
penultimo = numeros[-2]  # Obtiene el 4 (penúltimo elemento)
```

Es muy importante recordar que en Python (y en la mayoría de los lenguajes de programación), los índices comienzan en 0, no en 1. Un error común para los principiantes es intentar acceder al primer elemento con `lista[1]`, lo que en realidad accede al segundo elemento.

Modificando listas

Las listas son mutables, lo que significa que podemos cambiar sus elementos después de crearlas:

```
# Modificar elementos
numeros[0] = 10  # Ahora numeros es [10, 2, 3, 4, 5]

# Agregar elementos al final
numeros.append(6)  # Agrega al final: [10, 2, 3, 4, 5, 6]

# Insertar en una posición específica
numeros.insert(1, 15)  # Inserta 15 en la posición 1: [10, 15, 2, 3, 4, 5, 6]

# Eliminar elementos
numeros.pop()  # Elimina el último: [10, 15, 2, 3, 4, 5]
numeros.pop(1)  # Elimina el elemento en la posición 1: [10, 2, 3, 4, 5]
numeros.remove(10)  # Elimina el valor 10: [2, 3, 4, 5]

# Obtener la longitud de la lista
cantidad = len(numeros)  # Devuelve 4
```

Métodos importantes de las listas

Python ofrece varios métodos útiles para trabajar con listas:

Método	Descripción	Ejemplo
append()	Agrega un elemento al final	lista.append(10)
insert()	Inserta un elemento en una posición	lista.insert(0, "inicio")
remove()	Elimina la primera ocurrencia de un valor	lista.remove("Juan")
pop()	Elimina y devuelve el elemento en una posición	lista.pop() o lista.pop(3)
index()	Devuelve el índice de un elemento	lista.index("María")

count()	Cuenta ocurrencias de un elemento	lista.count(5)
sort()	Ordena la lista	lista.sort()
reverse()	Invierte el orden de los elementos	lista.reverse()

1.3 Operaciones comunes con listas

Una de las tareas más comunes en programación es recorrer todos los elementos de una lista para procesarlos. Hay varias formas de hacerlo:

Recorrido con un bucle for

```
# Recorrer una lista con for
frutas = ["manzana", "plátano", "naranja", "pera"]
for fruta in frutas:
    print(fruta)  # Imprime cada fruta en una línea separada
```

Este es el método más común y legible para recorrer una lista en Python. El bucle `for` automáticamente itera sobre cada elemento, asignándolo a la variable temporal que hemos llamado `fruta`.

Recorrido con índices

A veces necesitamos saber no solo el valor, sino también la posición de cada elemento:

```
# Recorrer con índices
for i in range(len(frutas)):
    print(f"Posición {i}: {frutas[i]}")
```

Este método es útil cuando necesitamos modificar elementos mientras recorremos la lista, o cuando la posición es relevante para nuestra lógica.

Rebanadas (slicing)

Las rebanadas nos permiten obtener sublistas de una lista original:

```
# Rebanadas (slicing) - formato: lista[inicio:fin]
# Nota: El índice 'fin' no se incluye en el resultado

primeras_dos = frutas[0:2]  # ["manzana", "plátano"]
ultimas_dos = frutas[-2:]   # ["naranja", "pera"]
copia_completa = frutas[:]  # Crea una copia de toda la lista

# También podemos especificar un "paso"
cada_segundo = frutas[::2]  # ["manzana", "naranja"]
lista_invertida = frutas[::-1]  # ["pera", "naranja", "plátano", "manzana"]
```

Ordenamiento

Python hace que ordenar listas sea muy sencillo:

```
# Ordenar listas
numeros = [5, 2, 8, 1, 9]
numeros.sort()  # Modifica la lista original: [1, 2, 5, 8, 9]
```

```
# Para obtener una copia ordenada sin modificar la original
numeros_originales = [5, 2, 8, 1, 9]
ordenados = sorted(numeros_originales)  # [1, 2, 5, 8, 9]

# Orden inverso
numeros.sort(reverse=True)  # [9, 8, 5, 2, 1]
```

Comprobación de pertenencia

Podemos verificar si un elemento existe en una lista:

```
# Verificar si un elemento existe
if "manzana" in frutas:
    print("Hay manzanas en la lista")

if "kiwi" not in frutas:
    print("No hay kiwis en la lista")
```

Esta operación es muy eficiente en Python y hace que el código sea más legible.

1.4 Listas anidadas (matrices)

Las listas pueden contener otras listas como elementos, lo que nos permite crear estructuras más complejas como matrices (arreglos bidimensionales):

```
# Lista de listas (matriz)
matriz = [
    [1, 2, 3],  # Primera fila
    [4, 5, 6],  # Segunda fila
    [7, 8, 9]   # Tercera fila
]
```

Para visualizarlo mejor:

```
matriz = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
```

Esto representa una matriz de 3x3:

```
1  2  3
4  5  6
7  8  9
```

Acceso a elementos en matrices

Para acceder a un elemento, necesitamos dos índices: el primero para la fila y el segundo para la columna:

```
# Acceder a elementos
elemento = matriz[1][2]  # Fila 1, columna 2: valor 6
```

```
# Modificar un elemento
matriz[0][0] = 10  # Cambia el primer elemento a 10
```

Recorrido de matrices

Para procesar todos los elementos de una matriz, usamos bucles anidados:

```
# Recorrer una matriz
for fila in matriz:
    for elemento in fila:
        print(elemento, end=" ")  # Imprime el elemento seguido de un espacio
    print()  # Salto de línea al final de cada fila
```

Este código imprimirá:

```
1 2 3
4 5 6
7 8 9
```

También podemos recorrer usando índices:

```
# Recorrido con índices
for i in range(len(matriz)):  # Para cada fila
    for j in range(len(matriz[i])):  # Para cada columna en esa fila
        print(f"matriz[{i}][{j}] = {matriz[i][j]}")
```

Las matrices son muy útiles para representar datos tabulares, imágenes (donde cada píxel tiene valores RGB), tableros de juegos (ajedrez, gato, etc.), y muchas otras aplicaciones.

2. Funciones

2.1 ¿Qué son las funciones?

Las funciones son uno de los conceptos más importantes en programación. Son bloques de código reutilizables que realizan una tarea específica cuando las llamamos. Puedes pensar en las funciones como pequeñas "máquinas" que:

1. Reciben datos de entrada (parámetros o argumentos)
2. Procesan esos datos siguiendo una lógica
3. Devuelven un resultado (valor de retorno)

En el mundo real, usamos el concepto de funciones constantemente. Por ejemplo, una licuadora es como una función: le das ingredientes (entradas), presionas un botón (la invocas), internamente hace un proceso, y te devuelve un jugo (salida).

¿Por qué son importantes las funciones?

- **Reutilización de código**: Escribes el código una vez y lo usas muchas veces
- **Modularidad**: Divides problemas complejos en partes más pequeñas y manejables
- **Mantenibilidad**: Es más fácil actualizar, corregir y mejorar tu código
- **Abstracción**: Ocultas la complejidad interna y te enfocas en el uso
- **Legibilidad**: Hacen que el código sea más organizado y comprensible

Imagina que tienes que calcular el área de varios círculos en tu programa. Sin funciones, tendrías que repetir la fórmula `π * radio²` cada vez. Con una función `calcular_area_circulo(radio)`, simplemente la llamas cuando la necesites.

2.2 Sintaxis básica

En Python, las funciones se definen usando la palabra clave `def`, seguida del nombre de la función y paréntesis que pueden contener parámetros:

```
# Definición de una función simple
def saludar():
    """Esta es una función simple que imprime un saludo."""
    print("¡Hola, mundo!")

# Llamada a la función
saludar()  # Imprime: ¡Hola, mundo!
```

Lo que está entre triple comillas `"""` es la documentación de la función (docstring), que explica qué hace la función. Es una buena práctica incluir docstrings en tus funciones.

Funciones con parámetros

Los parámetros son variables que reciben valores cuando se llama a la función:

```
# Función con parámetros
def saludar_persona(nombre):
    """Saluda a una persona por su nombre."""
    print(f"¡Hola, {nombre}!")

# Llamada con argumento
saludar_persona("Ignacio")  # Imprime: ¡Hola, Ignacio!
saludar_persona("Camila")   # Imprime: ¡Hola, Camila!
```

En este ejemplo, `nombre` es un parámetro de la función `saludar_persona`. Cuando llamamos a la función con `saludar_persona("Ignacio")`, el valor `"Ignacio"` se pasa como argumento y se asigna al parámetro `nombre`.

Funciones con valores de retorno

Las funciones pueden devolver valores usando la palabra clave `return`:

```
# Función con valor de retorno
def sumar(a, b):
    """Suma dos números y devuelve el resultado."""
    resultado = a + b
    return resultado

# Uso del valor retornado
total = sumar(5, 3)
print(f"La suma es: {total}")  # Imprime: La suma es: 8

# También podemos usar el resultado directamente
print(f"5 + 7 = {sumar(5, 7)}")  # Imprime: 5 + 7 = 12
```

Cuando una función llega a una instrucción `return`, deja de ejecutarse y devuelve el valor especificado. Si no hay instrucción `return`, la función devuelve `None` implícitamente.

2.3 Parámetros y argumentos

Python ofrece varias formas de pasar argumentos a las funciones:

Parámetros posicionales obligatorios

Estos son los parámetros estándar que deben proporcionarse en el mismo orden en que se definieron:

```
# Parámetros posicionales obligatorios
def describir_persona(nombre, edad):
    """Describe a una persona con su nombre y edad."""
    print(f"{nombre} tiene {edad} años")

describir_persona("Carolina", 25)  # Correcto: Carolina tiene 25 años
# describir_persona("Carlos")  # ¡Error! Falta el parámetro edad
```

Parámetros con valores por defecto

Podemos establecer valores predeterminados para los parámetros, que se utilizarán si no se proporciona un valor:

```
# Parámetros con valores por defecto
def saludar(nombre, mensaje="¡Hola!"):
    """Saluda a una persona con un mensaje personalizable."""
    print(f"{mensaje} {nombre}")

saludar("Luis")  # Usa el mensaje por defecto: ¡Hola! Luis
saludar("Ana", "¡Buen día!")  # Sobreescribe el valor por defecto: ¡Buen día! Ana
```

Los parámetros con valores por defecto deben venir después de los parámetros obligatorios.

Parámetros nominales

Podemos especificar a qué parámetro va cada argumento usando su nombre:

```
# Uso de parámetros nominales
describir_persona(edad=30, nombre="Roberto")  # Roberto tiene 30 años
```

Esto es especialmente útil cuando una función tiene muchos parámetros, ya que mejora la legibilidad y nos permite proporcionar solo los que queremos cambiar.

Número variable de argumentos

A veces, queremos que una función acepte cualquier cantidad de argumentos:

```
# Número variable de argumentos
def suma_total(*numeros):
    """Suma todos los números proporcionados."""
    total = 0
    for numero in numeros:
```

```
        total += numero
    return total

# Podemos llamar con cualquier cantidad de argumentos
resultado1 = suma_total(1, 2)  # Suma 1 + 2
resultado2 = suma_total(1, 2, 3, 4, 5)  # Suma 1 + 2 + 3 + 4 + 5
print(f"Resultado 1: {resultado1}")  # Imprime: Resultado 1: 3
print(f"Resultado 2: {resultado2}")  # Imprime: Resultado 2: 15
```

El asterisco `*` antes del parámetro `numeros` indica que puede recibir cualquier cantidad de argumentos posicionales, que se recopilarán en una tupla.

2.4 Alcance de variables (scope)

El alcance de una variable se refiere a la parte del programa donde la variable es accesible. Python tiene principalmente dos ámbitos:

Variables locales

Las variables definidas dentro de una función son locales a esa función y no son accesibles fuera de ella:

```
def calcular():
    # Esta variable solo existe dentro de la función
    resultado = 42
    print(f"Dentro de la función: {resultado}")
    return resultado

calcular()  # Imprime: Dentro de la función: 42
# print(resultado)  # ¡Error! 'resultado' no existe fuera de la función
```

Variables globales

Las variables definidas fuera de cualquier función son globales y se pueden acceder desde cualquier parte del programa:

```
# Variable global
contador = 0

def incrementar():
    # Para modificar una variable global, debemos declararla como global
    global contador
    contador += 1
    print(f"Contador dentro de la función: {contador}")

incrementar()  # Imprime: Contador dentro de la función: 1
print(f"Contador fuera de la función: {contador}")  # Imprime: Contador fuera de la función: 1
```

Sin la palabra clave `global`, Python crearía una nueva variable local llamada `contador` dentro de la función, dejando la variable global sin cambios.

Encadenamiento de alcances

Cuando Python busca una variable, primero la busca en el alcance local, luego en los alcances circundantes, y finalmente en el alcance global:

```
x = "global"

def externa():
    x = "externa"

    def interna():
        print(f"x en interna: {x}")  # Usa el x de 'externa'

    interna()
    print(f"x en externa: {x}")  # Usa su propio x

externa()  # Imprime: x en interna: externa \n x en externa: externa
print(f"x global: {x}")  # Imprime: x global: global
```

2.5 Funciones como objetos de primera clase

En Python, las funciones son objetos de primera clase, lo que significa que pueden ser:

1. Asignadas a variables
2. Pasadas como argumentos a otras funciones
3. Retornadas como resultados de otras funciones
4. Almacenadas en estructuras de datos como listas o diccionarios

Esta característica hace que Python sea muy flexible y poderoso:

```
# Asignar funciones a variables
def duplicar(x):
    return x * 2

operacion = duplicar  # Nota: sin paréntesis, no estamos llamando a la función
print(operacion(5))  # Imprime: 10
```

Funciones como argumentos

Podemos pasar funciones como argumentos a otras funciones:

```
# Funciones como argumentos
def aplicar_operacion(funcion, valor):
    """Aplica una función a un valor."""
    return funcion(valor)

def triplicar(x):
    return x * 3

resultado1 = aplicar_operacion(duplicar, 7)  # Aplica duplicar a 7
resultado2 = aplicar_operacion(triplicar, 7)  # Aplica triplicar a 7
print(resultado1)  # Imprime: 14
print(resultado2)  # Imprime: 21
```

Este concepto es fundamental para técnicas avanzadas como la programación funcional.

Funciones anónimas (lambda)

Las expresiones lambda son pequeñas funciones anónimas que pueden definirse en una sola línea:

```
# Funciones anónimas (lambda)
cuadrado = lambda x: x ** 2
print(cuadrado(4))  # Imprime: 16

# Útil en funciones que esperan otra función como argumento
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x**2, numeros))
print(cuadrados)  # Imprime: [1, 4, 9, 16, 25]
```

Las funciones lambda son ideales para operaciones simples, pero para lógica más compleja es mejor usar funciones definidas con `def`.

3. Ejemplos en casos reales

Ahora que hemos aprendido sobre arreglos y funciones, veamos cómo se utilizan en situaciones reales.

3.1 Gestión de datos con listas

Un caso de uso común es la gestión de datos, como un sistema de inventario. Veamos cómo podríamos implementarlo usando lo que hemos aprendido:

```
# Sistema básico de gestión de inventario
inventario = [
    {"producto": "Laptop", "precio": 800000, "stock": 15},
    {"producto": "Mouse", "precio": 15000, "stock": 50},
    {"producto": "Teclado", "precio": 25000, "stock": 35},
    {"producto": "Monitor", "precio": 150000, "stock": 20}
]
```

Aquí estamos usando una lista de diccionarios, donde cada diccionario representa un producto con sus atributos. Este es un patrón muy común en programación.

Ahora, implementemos algunas funciones para trabajar con este inventario:

```
def mostrar_inventario(inventario):
    """Muestra el inventario completo en un formato legible."""
    print("INVENTARIO ACTUAL:")
    print("-----------------")
    for item in inventario:
        print(f"Producto: {item['producto']}")
        print(f"Precio: ${item['precio']}")
        print(f"Stock: {item['stock']} unidades")
        print("-----------------")

def buscar_producto(inventario, nombre):
    """Busca un producto por nombre y devuelve su información."""
    for item in inventario:
        if item["producto"].lower() == nombre.lower():
```

```
            return item
    return None  # Retorna None si el producto no se encuentra

def valor_total_inventario(inventario):
    """Calcula el valor total del inventario (precio * stock)."""
    total = 0
    for item in inventario:
        total += item["precio"] * item["stock"]
    return total

# Uso de las funciones
mostrar_inventario(inventario)

# Buscar un producto
producto_buscado = buscar_producto(inventario, "mouse")
if producto_buscado:
    print(f"Producto encontrado: {producto_buscado['producto']} -
${producto_buscado['precio']}")
else:
    print("Producto no encontrado")

# Calcular valor total
total = valor_total_inventario(inventario)
print(f"El valor total del inventario es: ${total}")
```

Este ejemplo muestra cómo podemos crear funciones para:

1. Mostrar información de manera organizada
2. Buscar elementos específicos en una colección
3. Realizar cálculos basados en múltiples elementos

Aplicación en el mundo real:

Este tipo de sistema podría ser la base de un software de gestión de inventario para una tienda real. Podríamos expandirlo para incluir funcionalidades como:

- Agregar nuevos productos
- Actualizar precios o cantidades
- Registrar ventas y reducir el stock automáticamente
- Generar alertas cuando el stock esté bajo
- Calcular ganancias y estadísticas

3.2 Procesamiento de datos

Otro caso común es el procesamiento de datos, como el análisis de información meteorológica:

```
# Análisis de datos de temperatura
temperaturas_santiago = [
    [22, 20, 21, 23, 24, 23, 21],  # Semana 1
    [24, 25, 26, 25, 23, 21, 20],  # Semana 2
    [19, 18, 17, 20, 22, 24, 25],  # Semana 3
    [21, 22, 23, 24, 25, 25, 24]   # Semana 4
]
```

Esta es una matriz (lista de listas) donde cada lista interna representa las temperaturas diarias durante una semana.

Ahora implementemos funciones para analizar estos datos:

```
def promedio_semanal(datos):
    """Calcula el promedio de temperatura para cada semana."""
    promedios = []
    for semana in datos:
        promedio = sum(semana) / len(semana)
        promedios.append(round(promedio, 1))  # Redondear a 1 decimal
    return promedios

def temperatura_maxima(datos):
    """Encuentra la temperatura máxima de cada semana."""
    maximas = []
    for semana in datos:
        maximas.append(max(semana))
    return maximas

def dias_sobre_umbral(datos, umbral):
    """Cuenta los días con temperatura superior a un umbral."""
    contador = 0
    for semana in datos:
        for temp in semana:
            if temp > umbral:
                contador += 1
    return contador

# Uso de las funciones
promedios = promedio_semanal(temperaturas_santiago)
print(f"Promedio por semana: {promedios}")

maximas = temperatura_maxima(temperaturas_santiago)
print(f"Temperaturas máximas: {maximas}")

dias_calurosos = dias_sobre_umbral(temperaturas_santiago, 24)
print(f"Días con temperatura superior a 24°C: {dias_calurosos}")
```

Este ejemplo muestra cómo:

1. Procesar colecciones anidadas de datos
2. Realizar cálculos estadísticos básicos
3. Filtrar datos según criterios específicos

Aplicación en el mundo real:

Este tipo de procesamiento podría usarse en:

- Aplicaciones meteorológicas
- Análisis de datos científicos
- Seguimiento de indicadores económicos
- Monitoreo de sensores IoT (Internet de las Cosas)

3.3 Automatización de tareas

Las funciones y listas también son ideales para automatizar tareas repetitivas:

```
# Procesamiento automático de mensajes
mensajes = [
    "Recordar reunión mañana a las 10:00",
    "URGENTE: Revisar informe antes del viernes",
    "Feliz cumpleaños Juan!",
    "URGENTE: Actualizar sistema antes de las 18:00",
    "Recordar enviar documentos para revisión"
]
```

Implementemos funciones para procesar estos mensajes automáticamente:

```
def filtrar_mensajes(lista_mensajes, filtro):
    """Filtra mensajes que contienen una palabra clave específica."""
    return [mensaje for mensaje in lista_mensajes if filtro.lower() in
mensaje.lower()]

def priorizar_mensajes(lista_mensajes):
    """Separa mensajes en prioritarios y normales según si contienen 'URGENTE'."""
    prioritarios = []
    normales = []

    for mensaje in lista_mensajes:
        if "URGENTE" in mensaje:
            prioritarios.append(mensaje)
        else:
            normales.append(mensaje)

    return {"prioritarios": prioritarios, "normales": normales}

def resumen_mensajes(lista_mensajes):
    """Genera un resumen de la cantidad de mensajes por categoría."""
    cantidad = len(lista_mensajes)
    urgentes = len([m for m in lista_mensajes if "URGENTE" in m])
    recordatorios = len([m for m in lista_mensajes if "Recordar" in m])

    return {
        "total": cantidad,
        "urgentes": urgentes,
        "recordatorios": recordatorios,
        "otros": cantidad - urgentes - recordatorios
    }

# Uso de las funciones
mensajes_urgentes = filtrar_mensajes(mensajes, "urgente")
print("Mensajes urgentes:")
for mensaje in mensajes_urgentes:
    print(f"- {mensaje}")

clasificacion = priorizar_mensajes(mensajes)
print("\nMensajes prioritarios:", len(clasificacion["prioritarios"]))
```

```
print("Mensajes normales:", len(clasificacion["normales"]))

resumen = resumen_mensajes(mensajes)
print("\nResumen de mensajes:")
print(f"Total: {resumen['total']}")
print(f"Urgentes: {resumen['urgentes']}")
print(f"Recordatorios: {resumen['recordatorios']}")
print(f"Otros: {resumen['otros']}")
```

Este ejemplo demuestra:

1. Filtrado de datos según condiciones
2. Clasificación automática de información
3. Generación de resúmenes y estadísticas

Aplicación en el mundo real:

Estas técnicas podrían usarse para:

- Sistemas de correo electrónico para filtrar spam
- Asistentes virtuales para priorizar tareas
- Herramientas de análisis de redes sociales
- Sistemas de atención al cliente para clasificar consultas