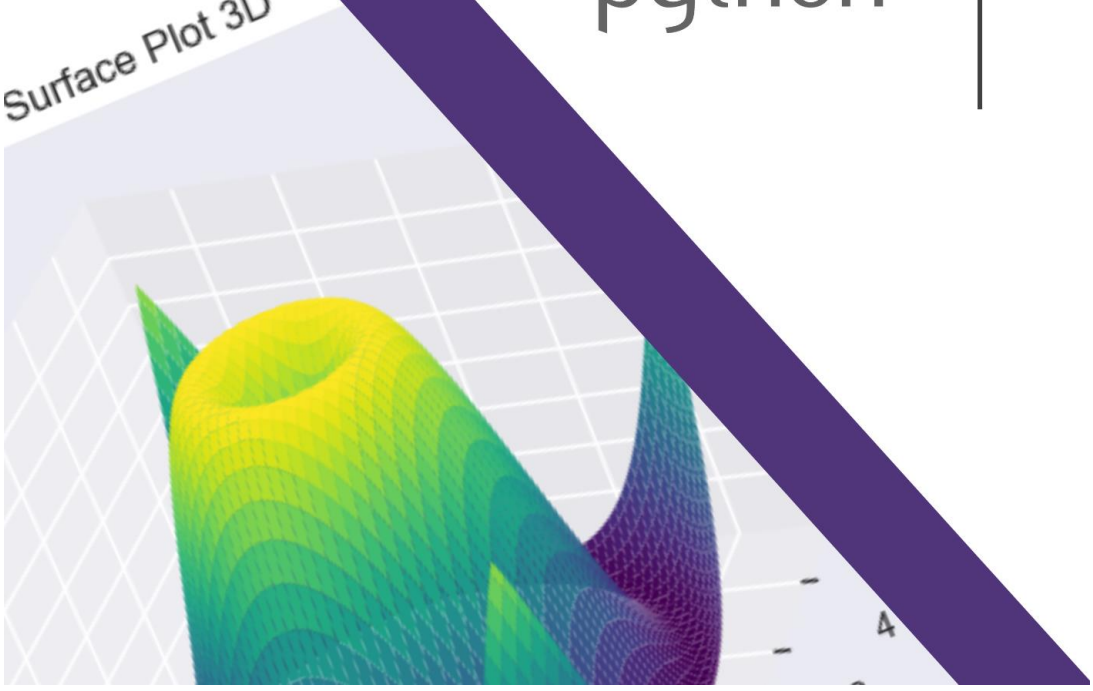
Francisco Alejandro Medina

INTRODUCCIÓN A LAS BIBLIOTECAS NUMPY Y MATPLOTLIB

python™



Introducción a las bibliotecas Numpy y Matplotlib de Python

FRANCISCO ALEJANDRO MEDINA AGUIRRE

UNIVERSIDAD TECNOLOGICA DE PERREIRA

FACULTAD DE INGENIERIAS

COLOMBIA, 2024

Tabla de contenido

LISTA DE FIGURAS

INTRODUCCION

En la era digital actual, donde el análisis de datos y la visualización se han convertido en habilidades esenciales en diversas disciplinas, las herramientas que permiten realizar estas tareas de manera eficiente son más valiosas que nunca. Python, conocido por su simplicidad y potencia, se destaca como uno de los lenguajes de programación más populares y versátiles, especialmente en el campo de la ciencia de datos y la ingeniería. Dentro de este ecosistema, las bibliotecas Numpy y Matplotlib emergen como pilares fundamentales para el cálculo numérico y la visualización de datos, respectivamente. Este documento tiene como objetivo proporcionar una guía exhaustiva sobre el uso de estas bibliotecas, enfatizando su aplicación práctica en problemas reales.

Numpy, abreviatura de Numerical Python, es la biblioteca por excelencia para la computación científica en Python. Ofrece una estructura de datos de alto rendimiento conocida como arrays, que permite realizar operaciones matemáticas complejas de manera eficiente. Su capacidad para manejar grandes volúmenes de datos, junto con una sintaxis intuitiva y una amplia gama de funciones matemáticas, la convierte en una herramienta indispensable para investigadores, científicos de datos, y cualquier profesional involucrado en el análisis cuantitativo. Este texto guía a los lectores a través de los fundamentos de Numpy, desde la creación y manipulación de arrays hasta la implementación de algoritmos matemáticos avanzados.

Por otro lado, Matplotlib se erige como la biblioteca líder en Python para la creación de visualizaciones estáticas, animadas e interactivas. Capaz de generar gráficos de alta calidad en una variedad de formatos, facilita la interpretación de datos complejos y la comunicación de hallazgos de manera clara y efectiva. A través de Matplotlib, este documento demuestra cómo transformar datos numéricos en representaciones visuales comprensibles, cubriendo desde gráficos básicos de líneas y barras hasta visualizaciones avanzadas personalizadas. La habilidad para visualizar datos correctamente es crucial no solo en la ciencia de datos, sino también en campos tan diversos como la economía, la biología, y la ingeniería.

Este texto está diseñado para principiantes que buscan una introducción sólida a estas bibliotecas, como para usuarios intermedios que desean profundizar su comprensión y ampliar sus habilidades en análisis y visualización de datos con Python. A través de

explicaciones breves, ejemplos de código prácticos y ejercicios aplicados, los lectores adquirirán una comprensión fundamental de Numpy y Matplotlib, preparándolos para enfrentar desafíos reales en sus campos de estudio o trabajo. Al final de este recorrido, se espera que los estudiantes y profesionales no solo sean competentes en el manejo de estas herramientas, sino que también sean capaces de aplicarlas creativamente para explorar y resolver problemas complejos.

CAPUTILO 1. FUNDAMENTOS DE PYTHON

Es uno de los lenguajes de programación de mayor crecimiento y popularidad en los últimos años. Fue creado a finales de la década de 1980 por el programador holandés Guido Van Rossum quien buscaba crear un lenguaje que fuera fácil de usar y aprender, incluso para no programadores.

Figura 1. Logo de lenguaje de programación Python

Python es multiparadigma, soporta programación orientada a objetos, imperativa y funcional. Se trata de un lenguaje interpretado, es decir que no necesita compilación y se ejecuta línea por línea. Esto permite ver resultados rápidamente sin procesos intermedios. También es multiplataforma, puede utilizarse en Windows, Mac, Linux y otros sistemas operativos.

La sintaxis de Python se caracteriza por ser muy legible y limpia, similar al inglés natural. Utiliza sangrías en lugar de corchetes para delimitar bloques de código, lo que le brinda mucho orden visual. También favorece la legibilidad el uso de nombres de variables explicativos. Asimismo, al ser un lenguaje de tipado dinámico, no es necesario declarar variables ni tipos de datos.

Python cuenta con una gran biblioteca estándar que incorpora muchas funciones útiles para distintas tareas de programación, evitando tener que reinventar la rueda. Además, tiene una amplia comunidad que desarrolla y mantiene bibliotecas o frameworks adicionales de código abierto para propósitos específicos, como ciencia de datos, web, robótica, inteligencia artificial, entre muchos otros.

La sintaxis sencilla de Python y su curva de aprendizaje suave lo convierten en un excelente primer lenguaje para aprender a programar. Permite enfocarse más en la lógica y estructuras

de programación antes que en los detalles del lenguaje. Además, Python es utilizado por principiantes como expertos dado que es un lenguaje muy versátil y potente.

Python es un lenguaje interpretado de propósito general, cuenta con tipado dinámico y una sintaxis clara, ordenada, de fácil lectura muy cercana al inglés. Posee además una gran biblioteca estándar y muchos frameworks de código abierto. Por todas estas características se ha convertido en uno de los lenguajes de programación más populares para iniciar en el aprendizaje como para desarrollar todo tipo de aplicaciones.

Desde scripts simples hasta complejos programas, Python es una excelente elección como primer lenguaje dado su sintaxis sencilla y enfoque práctico, que permiten concentrarse más en la lógica de programación. Cuenta con el poder y escalabilidad para crecer profesionalmente como programador. Es un lenguaje confiable utilizado por grandes compañías y aplicaciones web conocidas como Google, Youtube, Instagram, Spotify, entre muchas otras.

Python sigue mejorando y evolucionando constantemente. La versión estable más reciente es Python 3, que introdujo cambios y mejoras con respecto a Python 2 que ya no se mantiene. La comunidad Python es muy activa en agregar características y actualizaciones por medio de PEPs (Python Enhancement Proposals). El lenguaje se adapta así a las tendencias y necesidades cambiantes en el desarrollo de software.

Python se ha consolidado como uno de los lenguajes imperdibles y esenciales para todo desarrollador. Su creciente adopción se debe a características clave como legibilidad, versatilidad, facilidad de uso, amplia biblioteca, comunidad y ecoistema activo. Python tiene las cualidades ideales para introducirse en la programación y es un gran compañero de viaje para crecer como profesional. Aprender Python abre la puerta a un sinfín de posibilidades en el apasionante mundo de la tecnología.

Python es un lenguaje de programación multiparadigma muy popular por su facilidad de uso y sintaxis clara y legible. Fue creado por Guido van Rossum a principios de la década de 1990 con el objetivo de ser un lenguaje accesible para todo tipo de programadores, incluso principiantes.

Python es interpretado, dinámicamente tipado y multiplataforma. Cuenta con una amplia biblioteca estándar incorporada y una gran comunidad que desarrolla módulos y paquetes adicionales de código abierto para expandir sus capacidades en distintas áreas de aplicación como ciencia de datos, desarrollo web, sistemas operativos, entre muchas otras (Python Software Foundation, 2022).

Python se ha consolidado como uno de los lenguajes de programación más populares y versátiles de los últimos tiempos. Desde aplicaciones web, videojuegos, inteligencia artificial hasta administración de sistemas, Python tiene un amplio uso en la industria tecnológica actual (Downey, The importance of Python programming skills. In A. Downey (Ed.), Think Python, 2016).

Su creciente adopción se debe en gran parte a que cuenta con una curva de aprendizaje suave para principiantes pero con el poder de escalar a aplicaciones complejas. A continuación exploraremos los elementos fundamentales del lenguaje que todo programador Python debe conocer.

1.1. SINTAXIS BÁSICA.

La sintaxis de un lenguaje de programación comprende las reglas que gobiernan la estructura y elementos que componen los programas. En Python se destaca por priorizar la legibilidad y facilidad de escritura mediante el uso de sangrías e identaciones para delimitar bloques de código, en lugar de símbolos como paréntesis o llaves. Esto le brinda un formato visual muy ordenado y limpio (Sweigart, 2022).

Otra característica importante es que distingue entre mayúsculas y minúsculas (case sensitive), por lo que debemos mantener consistencia al nombrar variables y funciones. Los comentarios se indican con el símbolo de numeral (#), lo cual permite documentar y explicar diferentes partes del código sin afectar la ejecución. También se pueden hacer comentarios de varias líneas para explicar un bloque de código más grande, se inician con tres comillas simples (‘’’) o dobles (“””) al principio y al final del bloque.

Los programas de Python consisten en scripts o módulos almacenados en archivos con extensión “.py” que contienen las instrucciones interpretadas por el intérprete de Python.

La instrucción print es una de las más básicas y utilizadas en el lenguaje de programación Python, se usa para mostrar salidas de datos en la consola de comandos. Las salidas de datos pueden ser mensajes que se escriben entre comillas simples o dobles (Lutz, 2013).

Un ejemplo de un script simple es:

```
"""
   Este es un comentario de varias líneas
   Explica el funcionamiento de este bloque de código
"""
# Programa Hola Mundo
print('Hola Mundo') # Imprime string
print("Bienvenido a Python") # Otro string
nombre = "Juan" # Variable nombre
print(nombre) # Imprime variable
```

1.2 VARIABLES Y TIPOS DE DATOS.

Las variables son contenedores que almacenan valores o información para ser referenciados y manipulados en un programa. Python es un lenguaje dinámicamente tipado, lo que significa que no es necesario declarar explícitamente el tipo de dato al definir una variable, esto se determina automáticamente al asignarle un valor (Sweigart, 2022).

Python es sensible a las mayúsculas y minúsculas, lo cual quiere decir que distingue entre letras mayúsculas y minúsculas, por lo que debemos tener cuidado al nombrar variables, funciones, etc. Los nombres de variables no pueden iniciar con números, pero sí con guion bajo (_). Se recomienda utilizar la nomenclatura “snake_case” con guiones bajos para separar palabras en nombres de variables y funciones.

Los principales tipos de datos que se usarán en el código Python son:

- Enteros (int): para valores numéricos sin decimales. Ej: 25
- Flotantes (float): para valores numéricos con decimales. Ej: 3.1416
- Booleanos (bool): para valores True o False.
- Strings (str): para cadenas de texto. Ej: "Hola"
- Listas (list): colecciones de valores entre corchetes. Ej: [1, 2, 3]
- Tuplas (tuple): listas inmutables entre paréntesis. Ej: (1, True, "Hola")
- Diccionarios (dict): pares de clave-valor entre llaves. ej: {"nombre":"Juan","edad":20}

Un ejemplo de declaración de variables es:

```
# Variables en Python
entero = 15 # int
flotante = 10.5 # float
texto = "Bienvenido" # string
boleano = True # boolean
lista = [1, 2, 3] # list
tupla = ("a", "b", "c") # tuple
diccionario = {"nombre":"Ana", "edad":19} # dict
```

1.3. OPERADORES Y EXPRESIONES DE ASIGNACIÓN.

Los operadores son símbolos que representan acciones aplicables a valores y variables para realizar cálculos u operaciones. Los principales tipos de operadores en Python son (Downey, Think Python: How to think like a computer scientist (2nd ed.), 2015):

- Aritméticos: +, -, *, /, %, **, // para realizar operaciones matemáticas.
- Asignación: =, +=, -=, *=, /= para asignar valores.
- Comparación: ==, !=, >, <, >=, <= para comparar expresiones.
- Lógicos: and, or, not para combinar expresiones booleanas.
- Bitwise: &, |, ~, ^, >>, << para operar a nivel de bits.

Algunos ejemplos de uso:

```
# Operadores aritméticos
5 + 3 # Suma
10 - 4 # Resta
4 * 5 # Multiplicación
16 / 4 # División da un float
18 % 7 # Módulo o resto
5 ** 3 # Potencia
15 // 4 # División entera

# Operadores comparación
5 == 5 # Igualdad
10 != 5 # Desigualdad
15 > 10 # Mayor que

# Operadores lógicos
True and False # Y lógico
True or False # O lógico
not True # Negación

# Operadores bitwise
8 & 5 # AND a nivel de bits
```

```
9     | 3 # OR a nivel de bits
11^ 5 # XOR a nivel de bits
```

Estos operadores permiten realizar todo tipo de operaciones matemáticas, lógicas y a nivel de bits en Python. Se pueden combinar para crear expresiones un poco más complejas dentro del código, por ejemplo:

```
precio = 25
descuento = 12
precio_final = precio - (precio * descuento / 100)
print(precio_final) # Imprime 22.0
```

Python respeta las reglas convencionales de precedencia de operadores: paréntesis, exponenciación, multiplicación/división, suma/resta. Esto determina el orden en que se evalúan las expresiones para obtener el resultado correcto. Estos operadores permiten realizar todo tipo de manipulaciones y cálculos en un programa de Python según se necesite.

En Python existen diferentes operadores de asignación que nos permiten realizar una operación y asignación de forma simplificada en una sola expresión. El operador básico es el signo igual (=) que asigna el valor de la derecha a la variable de la izquierda (Sweigart, 2022):

```
x = 5 # asigna 5 a x
```

Además, están los operadores combinados como +=, -=, *=, /=, que combinan la operación con la asignación (Downey, 2015):

```
x += 3 # x = x + 3
x -= 2 # x = x - 2
x *= 4 # x = x * 4
x /= 2 # x = x / 2
```

Estos permiten acortar y simplificar expresiones donde queremos aplicar una operación a una variable existente.

También podemos encadenar múltiples asignaciones en una línea:

```
x = p = z = 0 # x, p y z asignados a 0
```

Donde los valores se asignan de derecha a izquierda.

En las asignaciones con expresiones podemos utilizar expresiones arbitrarias complejas del lado derecho:

```
x = (3 * 4) + (10 / 5)
```

Python evalúa la expresión, obtiene el resultado y luego lo asigna a la variable.

Las asignaciones múltiples permiten asignar varias variables en una sola expresión:

```
a, b, c = 10, 20, 30
```

Esto asigna 10 a ´a', 20 a 'b' y 30 a 'c' respectivamente.

Un uso muy práctico de la asignación múltiple es para intercambiar los valores de dos variables:

```
a = 10
b = 20
a, b = b, a # intercambia a y b
# a ahora es 20, b ahora es 10
```

las expresiones de asignación en Python nos permiten realizar operaciones y asignaciones simplificadas para escribir código más conciso y legible (Lutz, 2013), tienen varios usos para inicializar, modificar y manipular variables.

1.4 ESTRUCTURAS DE CONTROL.

Las estructuras de control de flujo permiten controlar el orden en que se ejecutan las instrucciones en un programa Python, de modo que podamos construir código más complejo y versátil que se adapte a distintas situaciones y requerimientos lógicos.

Python incluye varias estructuras de control que veremos a continuación:

1.4.1. Condicional if.

La sentencia if permite ejecutar código sólo si se cumple una condición booleana específica. Su sintaxis es:

```
if condicion:
  # código a ejecutar
```

Donde condicion puede ser cualquier expresión que evalúe a True o False. Por ejemplo:

```
x = 10
if x > 5:
  print("x es mayor a 5")
```

Esto imprimirá el mensaje porque la condición x > 5 se cumple.

Podemos encadenar múltiples condiciones con elif (abreviación de else if):

```
color = "rojo"
if color == "verde":
      print("El color es verde")
elif color == "rojo":
      print("El color es rojo")
elif color == "azul":
      print("El color es azul")
else:
      print("Color no reconocido")
```

Si ninguna condición if/elif se cumple, se ejecuta el bloque else opcional.

1.4.2. Ciclo for.

El ciclo for itera sobre los elementos de una secuencia que se repite, ejecutando el código en cada iteración. Por ejemplo, para iterar sobre una lista:

```
frutas = ["manzana", "banana", "kiwi"]
for fruta in frutas:
      print("La fruta es:", fruta)
```

O sobre un rango numérico con range():

```
for i in range(5):
   print(f"i vale {i}")
```

Esto imprime los números del 0 al 4.

También podemos iterar sobre caracteres de un string con for:

```
for letra in "Python":
   print(letra)
```

1.4.3. Ciclo while.

El ciclo while ejecuta un bloque de código mientras se cumpla una condición booleana. Por ejemplo:

```
i = 0
while i < 5:
   print(f"i vale {i}")
   i += 1 # incrementa i
```

Esto imprime el valor de 'i' mientras 'i' sea menor que 5, actualizando la variable en cada iteración.

1.4.4. Instrucciones break, continue, pass.

Estas instrucciones nos brindan mayor control sobre los ciclos:

- break: Termina completamente el ciclo actual
- continue: Vuelve al comienzo del ciclo
- pass: Equivale a no hacer nada, útil como marcado de posición o placeholder ósea un espacio en blanco o un símbolo que indica dónde se insertará una información particular más adelante

Por ejemplo:

```
# Rompe el ciclo si i es 3
for i in range(10):
  if i == 3:
    break
  print(i)

# Pasa a la siguiente iteración si i es impar
for i in range(10):
  if i % 2 != 0:
    continue
  print(i)

# Hace "nada"
for i in range(10):
  pass # placeholders
```

Dominar el uso de estructuras de control en Python nos permite crear programas versátiles capaces de hacer frente a todo tipo de situaciones y requerimientos lógicos. Estas construcciones de flujo son parte integral de cualquier lenguaje de programación.

1.5. FUNCIONES.

Las funciones son una característica esencial en cualquier lenguaje de programación. Una función agrupa un bloque de código para realizar una tarea o cálculo específico, lo cual promueve la reutilización, organización y modularidad del código (Lutz, 2013).

En Python podemos reconocer fácilmente las funciones porque su nombre va seguido de paréntesis. Veamos en detalle el concepto, sintaxis, tipos y usos de funciones en Python.

¿Qué es una función? Una función es un bloque de código identificado por un nombre que puede ser ejecutado invocando dicho nombre. Permite encapsular una secuencia de sentencias para realizar una tarea específica.

Las funciones pueden recibir entrada en forma de argumentos y parámetros, procesarlos de algún modo, y devolver una salida o resultado. Esto permite reutilizar las mismas instrucciones repetidamente sin tener que escribir el código desde cero (Sweigart, 2022).

1.5.1. Sintaxis de funciones.

La sintaxis básica para definir una función en Python es:

```
def nombre_funcion(parametros):
  # cuerpo de la función
  return resultado # opcional
```

Donde:

- def indica que se está definiendo una función.
- nombre_funcion es el identificador de la función.
- parámetros son los nombres para los datos de entrada. Son opcionales.
- return devuelve un resultado opcional al terminar.

Para ejecutar o llamar a la función se escribe su nombre seguido de paréntesis ():

```
nombre_funcion() # ejecuta la función
```

1.5.2. Parametrizando funciones.

Podemos definir funciones que reciban parámetros o argumentos como entrada. Estos se incluyen entre paréntesis en la definición:

```
def suma(a, b):
  total = a + b
  return total
```

```
resultado = suma(5, 8) # pasa 5 y 8 a los parámetros
print(resultado) # Imprime 13
```

Los parámetros actúan como variables locales dentro de la función.

1.5.3. Valores por defecto.

Podemos asignar valores por defecto a los parámetros para cuando no se pasen argumentos:

```
def saludar(nombre="Amigo"):
  print(f"Hola {nombre}")

saludar("Ana") # Salida: Hola Ana
saludar() # Salida: Hola Amigo
```

Esto hace la función más flexible y reusable.

1.5.4. Devolviendo valores.

Con return podemos devolver o retornar un valor desde la función:

```
def suma(a, b):
  return a + b

resultado = suma(10, 5) # se asigna el return a resultado
```

Si no hay return, la función devuelve None por defecto.

1.5.5. Tipos de funciones Python.

En Python encontramos distintos tipos de funciones:

- Funciones integradas: ya vienen incorporadas con Python como print(), len(), int(), etc.
- Módulos de la librería estándar: funciones agrupadas para realizar tareas específicas, como matemáticas, entrada/salida, web, OS, etc.
- Funciones definidas por el usuario: las que nosotros mismos creamos en nuestros programas para reutilizar código.
- Funciones anónimas o lambda: funciones sin nombre creadas en una sola línea con la palabra reservada lambda.
- Funciones recursivas: que se llaman a sí mismas para dividir problemas grandes en partes más pequeñas.

Veamos algunos ejemplos prácticos del uso de funciones en Python:

- Función para calcular el cubo de un número:

```
def cubicar(x):
  return x ** 3

resultado = cubicar(3)
print(resultado) # 27
```

- Función para saludar pasando nombre:

```
def saludar(nombre):
      print(f"¡Hola, {nombre}!")

saludar("Ana") # ¡Hola, Ana!
saludar("Pedro") # ¡Hola, Pedro!
```

- Funciones recursivas: Las funciones recursivas son un tipo de función en Python que se caracteriza por llamarse a sí misma dentro de su definición. Esto permite dividir problemas complejos en problemas más pequeños cada vez, hasta llegar a un caso base sencillo. (Downey, Think Python: How to think like a computer scientist (2nd ed.), 2015)

  La recursividad es una técnica muy útil para problemas que se pueden descomponer naturalmente en partes más simples del mismo tipo. Un ejemplo clásico son los algoritmos para calcular números factoriales:

```
def factorial(x):
  if x == 1:
    return 1
  else:
    return x * factorial(x-1)

print(factorial(5)) # 120
```

  Analicemos este ejemplo:

  - La función factorial() se llama a sí misma dentro del else.
  - El caso base es cuando x == 1, ahí se devuelve 1 y se termina la recursión.
  - La llamada recursiva es factorial(x - 1) que reduce el problema en 1 cada vez.

  De esta forma se calcula:

  - factorial(5) = 5 * factorial(4)
  - factorial(4) = 4 * factorial(3)
  - factorial(3) = 3 * factorial(2)
  - factorial(2) = 2 * factorial(1)
  - factorial(1) = 1 (caso base)

Y se resuelve el problema combinando las llamadas recursivas.

La recursión requiere:

- Un caso base que termine la recursión.
- Una llamada recursiva que acerque al caso base.

Ventajas:

- Soluciones elegantes y simples para problemas complejos.
- Código más corto y fácil de entender que iteraciones.

Desventajas:

- Alto consumo de memoria por las múltiples llamadas.
- Riesgo de recursión infinita si no hay caso base.

la recursión es una técnica avanzada que con práctica permite escribir algoritmos muy eficientes en Python. Debe usarse con precaución para evitar problemas de rendimiento o infinitos.

- Función lambda para sumar dos números:

```
sumar = lambda x, y: x + y

resultado = sumar(10, 15)
print(resultado) # 25
```

las funciones son bloques de código reutilizables que nos permiten crear programas Python más modulares, organizados y fáciles de mantener. Existen muchos tipos de funciones que podemos aprovechar en nuestros proyectos.

1.6. CLASES Y OBJETOS.

Python es un lenguaje que soporta ampliamente la programación orientada a objetos mediante clases y objetos. Esta es una forma muy útil y poderosa de organizar y diseñar nuestros programas.

1.6.1. ¿Qué es una clase?

Una clase es como un plano o blueprint para crear objetos. Define un conjunto de atributos (datos) y métodos (funciones) que serán compartidos por todos los objetos creados a partir de esa clase (Lott, 2022).

Las clases nos permiten modelar entidades del mundo real, como personas, vehículos, cuentas bancarias, etc. Cada objeto creado a partir de la clase se denomina instancia y contiene sus propios datos.

La sintaxis para definir una clase en Python es:

```
class NombreClase:

  # atributos de clase

  def __init__(self):
    # constructor initializer

  # métodos de clase

  def nombre_metodo(self):
    # código del método
```

Donde __init__() es el constructor que se ejecuta al crear un objeto. self hace referencia al objeto actual.

1.6.2. Creando objetos.

Podemos crear instancias u objetos a partir de una clase usando la notación:

```
obj1 = NombreClase() # crea objeto 1
obj2 = NombreClase() # crea objeto 2
```

Cada objeto tendrá sus propios atributos y compartirán los mismos métodos de clase.

1.6.3. Atributos.

Los atributos almacenan los datos asociados a un objeto particular. Se definen dentro de la clase y se accede con la notación punto:

```
obj1.atributo = valor # asignar valor
valor = obj1.atributo # acceder valor
```

1.6.4. Métodos.

Los métodos son funciones que representan comportamientos de los objetos. Se definen en la clase y se acceden también con notación punto:

```
obj1.metodo() # llamar método
```

Dentro de los métodos se utiliza self para referir los atributos del objeto actual.

Ejemplo de clase Persona: Veamos un ejemplo de una clase Persona con atributos nombre y edad, y métodos presentarse y cumplir_años():

```
class Persona:

  def __init__(self, nombre, edad):
    self.nombre = nombre # atributo
    self.edad = edad # atributo

  def presentarse(self):
    print(f"Hola soy {self.nombre}, tengo {self.edad} años")

  def cumplir_anos(self):
    self.edad += 1

# crear objetos
persona1 = Persona("Juan", 30)
persona2 = Persona("Ana", 18)

# llamar métodos
persona1.presentarse() # Hola soy Juan, tengo 30 años
persona2.cumplir_anos() # actualiza edad a 19
```

Esto demuestra los conceptos centrales de programación orientada a objetos en Python.

Una característica poderosa de manejar objetos es la herencia entre clases. Esto permite crear clases que hereden atributos y métodos de una clase padre, pudiendo definir comportamiento propio, la herencia proporciona reutilización de código y modelado de relaciones entre entidades del mundo real. La clase hija se define indicando entre paréntesis la clase padre de la que hereda.

las clases y objetos en Python nos permiten crear programas mejor estructurados, reutilizables y fáciles de mantener. Es un paradigma fundamental para desarrollar sistemas complejos de forma organizada y eficiente.

1.7. BIBLIOTECAS EN PYTHON.

Una biblioteca en Python es un conjunto de funciones y clases que se pueden utilizar en un programa. Las bibliotecas se pueden encontrar en línea o en el repositorio de paquetes de Python (PyPI).

Para utilizar una biblioteca en Python, primero debemos importarla. Esto se puede hacer usando el comando import. Por ejemplo, para importar la biblioteca math, podemos usar el siguiente código:

```
import math
```

Una vez que hemos importado una biblioteca, podemos acceder a sus funciones y clases usando el operador punto (.). Por ejemplo, para calcular el valor de pi, podemos usar el siguiente código:

```
import math
pi = math.pi
print(pi)
```

1.7.1. Tipos de bibliotecas en Python

Las bibliotecas en Python se pueden clasificar en dos tipos principales:

- **Bibliotecas estándar:** Estas bibliotecas forman parte del lenguaje Python, y están disponibles de forma predeterminada.
- **Bibliotecas de terceros:** Estas bibliotecas son desarrolladas por terceros, y deben instalarse por separado.

1.7.1.1. Bibliotecas estándar de Python.

La biblioteca estándar de Python incluye una amplia gama de funciones y clases que pueden utilizarse para realizar tareas comunes. Algunas de las bibliotecas estándar más importantes son:

- **Biblioteca** math**:** Esta biblioteca proporciona funciones para el cálculo numérico, como el cálculo de raíces cuadradas, logaritmos y funciones trigonométricas.

- **Biblioteca** random**:** Esta biblioteca proporciona funciones para generar números aleatorios.
- **Biblioteca** datetime**:** Esta biblioteca proporciona funciones para trabajar con fechas y horas.
- **Biblioteca** os**:** Esta biblioteca proporciona funciones para trabajar con el sistema operativo.
- **Biblioteca** sys**:** Esta biblioteca proporciona funciones para acceder a información sobre el sistema.

1.7.1.2. Bibliotecas de terceros en Python

Existen muchas bibliotecas de terceros disponibles para Python. Estas bibliotecas suelen ser elaboradas por comunidades de desarrolladores, y proporcionan funciones y clases especializadas para tareas específicas. Algunas de las bibliotecas de terceros más populares son:

- **Biblioteca** numpy**:** Esta biblioteca proporciona funciones para el cálculo numérico de alto rendimiento.
- **Biblioteca** scipy**:** Esta biblioteca proporciona funciones para el análisis estadístico y científico.
- **Bibliotecas pandas:** Esta biblioteca proporciona funciones para trabajar con datos tabulares.
- **Biblioteca** matplotlib**:** Esta biblioteca proporciona funciones para la visualización de datos.
- **Biblioteca** flask**:** Esta biblioteca proporciona un marco web para el desarrollo de aplicaciones web.

1.7.2. Cómo instalar bibliotecas de terceros en Python

Para instalar una biblioteca de terceros en Python, podemos usar el administrador de paquetes de Python (pip). Pip es una herramienta que nos permite instalar y administrar paquetes de Python, para instalar una biblioteca de terceros usando pip, podemos usar el siguiente comando:

- pip install [nombre_de_la_biblioteca]

Por ejemplo, para instalar la biblioteca numpy, podemos usar el siguiente comando:

- pip install numpy

1.7.3. Ventajas de las bibliotecas en Python.

Las bibliotecas en Python ofrecen una serie de ventajas, entre las que se incluyen:

- **Reutilización de código:** Las bibliotecas nos permiten reutilizar código que ya ha sido desarrollado y probado por otros desarrolladores. Esto nos ahorra tiempo y esfuerzo.
- **Especialización:** Las bibliotecas proporcionan funciones y clases especializadas para tareas específicas. Esto nos permite centrarnos en la lógica de nuestro programa, sin tener que preocuparnos por implementar las funciones y clases básicas.
- **Documentación:** Las bibliotecas suelen estar bien documentadas, lo que nos facilita su uso.

Las bibliotecas en Python son una herramienta esencial para cualquier programador de Python, nos permiten reutilizar código, especializarnos en tareas específicas y ahorrar tiempo y esfuerzo.

CAPUTILO 2. LIBRERÍA NUMPY

NumPy, acrónimo de Numerical Python, es una biblioteca esencial para la computación científica en Python, ofreciendo soporte robusto para grandes arrays y matrices, junto con un vasto conjunto de funciones matemáticas para operar sobre estos datos (Oliphant, 2006). Este recorrido histórico destaca cómo NumPy se ha convertido en un pilar para análisis de datos, ciencia de datos y computación científica, marcando una evolución significativa desde sus orígenes.

La génesis de NumPy se encuentra en la última parte de la década de 1990, cuando la necesidad de una herramienta eficiente para cálculos numéricos en Python se volvió evidente. Numeric, también conocido como Numerical Python, fue el primer intento de ofrecer dicha funcionalidad, desarrollado por Jim Hugunin. Aunque Numeric marcó un hito importante, presentaba limitaciones en términos de rendimiento y flexibilidad. En respuesta a estas limitaciones, el proyecto Numarray fue lanzado, ofreciendo mejoras en el manejo de arrays de gran tamaño pero sacrificando eficiencia para arrays menores. La existencia de dos paquetes fragmentó la comunidad Python, resaltando la necesidad de una solución unificada (Oliphant, 2006).

Travis Oliphant, reconociendo la necesidad de consolidar las características de Numeric y Numarray, lideró el esfuerzo para fusionar ambos en una sola biblioteca: NumPy. En 2005, Oliphant lanzó NumPy 1.0, estableciendo un nuevo estándar para el cálculo numérico en Python (Oliphant, 2006). Esta unificación no solo resolvió la fragmentación de la comunidad sino que también optimizó el rendimiento y expandió las capacidades matemáticas disponibles para los usuarios de Python.

Desde su creación, NumPy ha experimentado una evolución constante, impulsada por una comunidad de desarrolladores activa y comprometida. Ha mejorado en rendimiento, funcionalidad y accesibilidad, convirtiéndose en el fundamento sobre el cual se construyen otras bibliotecas esenciales en el ecosistema de ciencia de datos de Python, como Pandas, SciPy, Matplotlib y scikit-learn (VanderPlas, 2016).

A diferencia de MATLAB o R, NumPy se integra profundamente con Python, ofreciendo una flexibilidad incomparable para el análisis y visualización de datos, al tiempo que permite un desarrollo más ágil y adaptativo. La naturaleza de código abierto de NumPy, a diferencia de las licencias pagadas requeridas por herramientas como MATLAB, democratiza el acceso a herramientas poderosas de cálculo numérico, promoviendo una innovación y colaboración más amplias dentro de la comunidad científica y tecnológica.

NumPy ha transformado el panorama del análisis numérico y la ciencia de datos, proporcionando una herramienta potente y flexible para la manipulación de datos y cálculos matemáticos. Su desarrollo desde una necesidad de funcionalidad numérica hasta convertirse en una pieza central de la computación científica en Python ejemplifica la evolución de la programación científica y el análisis de datos.

2.1. LOS ARRAYS EN NUMPY.

NumPy, una abreviatura de Numerical Python, es una biblioteca esencial en el mundo de la ciencia de datos y la computación científica, principalmente debido a su estructura de datos estrella: el array. A diferencia de las listas tradicionales de Python, los arrays de NumPy están diseñados específicamente para realizar operaciones numéricas de manera eficiente.

Un array de NumPy es una cuadrícula de valores, todos del mismo tipo, y es indexado por una tupla de enteros no negativos. La cantidad de dimensiones es el rango del array; la forma de un array es una tupla de enteros dando el tamaño del array a lo largo de cada dimensión (VanderPlas, 2016).

2.1.1 Características Clave de los Arrays de NumPy

- Homogeneidad de Tipo: A diferencia de las listas de Python, todos los elementos en un array NumPy deben ser del mismo tipo, lo que optimiza el almacenamiento y el procesamiento de datos.
- Operaciones Vectorizadas: Los arrays permiten operaciones vectorizadas, lo que significa que las operaciones se realizan elemento por elemento, permitiendo un cálculo numérico eficiente y rápido.

- Multidimensionalidad: NumPy soporta arrays de una dimensión (vectores), dos dimensiones (matrices) y n dimensiones, facilitando la realización de cálculos complejos en datos de alta dimensionalidad.

2.1.2. Ventajas de los Arrays de NumPy sobre las Listas Tradicionales de Python

- Eficiencia en el Uso de Memoria: Los arrays de NumPy utilizan mucho menos memoria que las listas tradicionales de Python, ya que NumPy asigna un bloque contiguo de memoria, reduciendo la sobrecarga.
- Rendimiento Mejorado: Al estar implementados en C y utilizar operaciones vectorizadas, los arrays de NumPy son notablemente más rápidos que las listas de Python, especialmente para grandes volúmenes de datos y operaciones matemáticas complejas.
- Funcionalidad Avanzada: NumPy ofrece una amplia gama de funciones matemáticas y estadísticas optimizadas para trabajar con arrays, lo que no está disponible directamente con las listas de Python.

2.1.3. Ejemplos de Operaciones Básicas con Arrays

- Creación de Arrays:

```
import numpy as np
a = np.array([1, 2, 3])  # Crea un array de una dimensión
```

- Operaciones Matemáticas:

```
b = np.array([4, 5, 6])
suma = a + b  # Suma elemento a elemento
```

- Indexación y Slicing:

```
primer_elemento = a[0]  # Obtiene el primer elemento
subarray = a[0:2]  # Slicing: obtiene los dos primeros elementos
```

- Cambio de Forma:

```
c = np.array([[1, 2, 3], [4, 5, 6]])
c.reshape(3, 2)  # Cambia la forma del array a 3 filas y 2 columnas
```

Estos ejemplos ilustran cómo las operaciones básicas con arrays de NumPy son tanto intuitivas como poderosas, permitiendo a los usuarios manipular datos numéricos con facilidad.

2.2. EFICIENCIA Y RENDIMIENTO DE NUMPY

La eficiencia de NumPy se debe principalmente a su implementación interna y estructura de datos. Los arrays de NumPy son almacenados en bloques contiguos de memoria, a diferencia de las listas de Python, que son arrays de punteros a objetos dispersos en la memoria. Esta contigüidad permite que operaciones que serían computacionalmente costosas en listas de Python se realicen de manera mucho más eficiente en NumPy, aprovechando la vectorización y operaciones realizadas a nivel de compilador optimizado en C.

Además, NumPy realiza operaciones en "bloques", lo que minimiza el número de ciclos de interpretación de Python requeridos. Esto se conoce como "Universal Functions" (ufuncs) en NumPy, permitiendo que operaciones que normalmente requerirían bucles en Python se realicen en código compilado de alta velocidad (VanderPlas, 2016).

En el análisis de datos, especialmente con grandes conjuntos de datos, NumPy demuestra ser invaluable. Por ejemplo, al realizar operaciones de agregación como la suma, media o desviación estándar en conjuntos de datos grandes, NumPy puede realizar estos cálculos en una fracción del tiempo que tomarían las operaciones equivalentes en listas de Python puro o ciclos for.

Consideremos un conjunto de datos de un millón de números generados aleatoriamente. Con NumPy, calcular la media de estos datos es tan simple y rápido como np.mean(data), donde data es un array de NumPy. En comparación, realizar esta operación en una lista de Python requeriría un bucle for, que sería significativamente más lento.

Un estudio de caso relevante es el uso de NumPy en el campo de la bioinformática, donde el manejo y análisis de grandes conjuntos de datos genéticos son críticos. NumPy permite a los investigadores realizar análisis complejos, como la comparación de secuencias genéticas o la simulación de poblaciones genéticas, de manera eficiente. Otro estudio de caso es en el ámbito de la física computacional, donde NumPy se utiliza para simular sistemas físicos complejos, como flujos de fluidos y dinámicas de partículas, que requieren la manipulación de grandes matrices y vectores.

NumPy es una herramienta poderosa que ofrece mejoras significativas en el rendimiento y la eficiencia para el análisis de datos y la computación científica en Python. Su diseño y optimizaciones permiten a los científicos de datos y investigadores manejar grandes volúmenes de datos de manera efectiva, realizando cálculos complejos en tiempos que serían inalcanzables con las estructuras de datos estándar de Python.

2.3. APLICACIONES DE NUMPY EN DIVERSOS CAMPOS

NumPy, una de las bibliotecas más fundamentales en el ecosistema de Python para la computación científica, ha encontrado aplicaciones en una amplia gama de disciplinas científicas y de ingeniería. Su capacidad para realizar operaciones matemáticas y estadísticas de manera eficiente lo convierte en una herramienta indispensable para profesionales y académicos en campos tan variados como la física, ingeniería y bioinformática.

En el campo de la física, NumPy se utiliza para simular y modelar fenómenos complejos. Por ejemplo, en la mecánica cuántica y la relatividad general, donde el manejo de grandes matrices y vectores es común, NumPy facilita la implementación de algoritmos que resuelven ecuaciones diferenciales y sistemas lineales complejos. Un ejemplo notable es el uso de NumPy para simular sistemas de partículas en dinámica molecular, donde se calculan las trayectorias de partículas basadas en interacciones físicas (VanderPlas, 2016).

En ingeniería, NumPy es fundamental para el análisis de datos y la modelización numérica. Desde el diseño de estructuras hasta la simulación de sistemas eléctricos, NumPy permite a los ingenieros realizar cálculos precisos de elementos finitos y optimización. Además, se utiliza en el procesamiento de señales y la imagenología, donde las operaciones sobre arrays facilitan el filtrado, la transformación y la visualización de señales e imágenes.

La bioinformática, que combina biología, ciencias de la computación y estadísticas, utiliza NumPy para analizar y procesar grandes conjuntos de datos genéticos y proteómicos. NumPy permite el manejo eficiente de secuencias de ADN y la realización de cálculos estadísticos en estudios de asociación genómica amplia (GWAS), así como en la modelización de estructuras moleculares y la simulación de reacciones bioquímicas.

2.4. INTEGRACIÓN DE NUMPY CON OTRAS BIBLIOTECAS DE PYTHON

Una de las mayores fortalezas de NumPy es su capacidad para integrarse con otras bibliotecas y herramientas en el ecosistema de Python, creando un entorno poderoso y flexible para la ciencia de datos y la computación científica.

- SciPy: Construido sobre NumPy, SciPy es una biblioteca para matemáticas, ciencia e ingeniería que expande las capacidades de cálculo de NumPy con funciones adicionales para optimización, integración, interpolación, autovalores, estadísticas, y más.
- Pandas: Para análisis de datos y modelado estadístico, Pandas se integra con NumPy para ofrecer estructuras de datos y herramientas de manipulación de datos de alto nivel y rendimiento, como DataFrames, que son esenciales para el análisis de datos estructurados.
- Matplotlib: En visualización de datos, Matplotlib trabaja conjuntamente con NumPy para ofrecer una amplia gama de herramientas gráficas para representar datos numéricos en formatos visuales comprensibles, desde histogramas hasta gráficos 3D.
- Scikit-learn: En el aprendizaje automático, Scikit-learn utiliza NumPy para el manejo de datos y operaciones matemáticas en algoritmos de clasificación, regresión, clustering y reducción de dimensionalidad.

La versatilidad y eficiencia de NumPy lo han convertido en una piedra angular en el análisis de datos y la computación científica en Python. Su uso en campos como la física, ingeniería y bioinformática subraya su importancia en la investigación científica y el desarrollo tecnológico. La capacidad de NumPy para integrarse con otras bibliotecas amplía aún más su utilidad, facilitando un flujo de trabajo cohesivo y eficiente para científicos de datos, investigadores y profesionales de la ingeniería en todo el mundo. Esta integración sin fisuras entre NumPy y otras herramientas clave en Python asegura que los usuarios puedan abordar problemas complejos de análisis de datos con una suite de herramientas coherente y potente.

La colaboración entre NumPy y estas bibliotecas no solo mejora la eficiencia del flujo de trabajo sino que también promueve la innovación al permitir a los usuarios centrarse en la solución de problemas complejos en lugar de preocuparse por los detalles de implementación de bajo nivel. Esta sinergia es especialmente evidente en proyectos

interdisciplinarios, donde la combinación de análisis numérico, procesamiento de señales, modelado estadístico y visualización de datos se vuelve crucial.

2.5. CASOS DE ESTUDIO

Para ilustrar la eficiencia de NumPy en la práctica, consideremos algunos estudios de caso específicos:

- Astronomía: En proyectos como el [Large Synoptic Survey Telescope](https://www.lsst.org), NumPy ha sido utilizado para procesar imágenes del cielo nocturno, ayudando a identificar objetos celestes y fenómenos astronómicos. La capacidad de NumPy para manejar grandes arrays de datos ha sido fundamental para el análisis y procesamiento de estas imágenes.
- Neurociencia: En el campo de la neurociencia computacional, herramientas como NumPy facilitan el análisis de datos de electroencefalografía (EEG) y resonancia magnética funcional (fMRI), permitiendo a los investigadores estudiar la actividad cerebral con un nivel de detalle sin precedentes.
- Finanzas Cuantitativas: NumPy también se utiliza en finanzas cuantitativas para modelar y simular opciones financieras y riesgos de mercado. La eficiencia de NumPy en cálculos matemáticos complejos permite a los analistas realizar simulaciones Monte Carlo y análisis de series temporales en grandes conjuntos de datos financieros.

La eficiencia y versatilidad de NumPy, combinadas con su integración con otras bibliotecas de Python, lo convierten en una herramienta indispensable en la ciencia de datos y la computación científica. Desde la simulación de fenómenos físicos hasta el análisis de datos genómicos y la visualización de complejas estructuras financieras, NumPy facilita el camino para descubrimientos científicos y avances tecnológicos. A medida que el mundo se vuelve cada vez más guiado por datos, herramientas como NumPy serán cruciales para analizar, interpretar y visualizar el vasto océano de datos generados en todas las esferas de la investigación y la industria.

NumPy, una de las bibliotecas más fundamentales para la computación científica en Python, juega un papel crucial en el análisis de datos y el aprendizaje automático. Gracias a su eficiencia en el manejo de arrays y matrices, NumPy facilita no solo el análisis de grandes volúmenes de datos sino también la implementación de algoritmos complejos de machine learning.

NumPy ofrece una amplia gama de operaciones matemáticas y estadísticas que pueden aplicarse directamente a arrays y matrices, lo que es esencial para el análisis de datos y el desarrollo de modelos de machine learning. Sus capacidades para realizar operaciones a nivel de array permiten el procesamiento de datos a alta velocidad, una necesidad en el análisis de grandes conjuntos de datos. Además, NumPy se integra perfectamente con otras bibliotecas de Python especializadas en análisis de datos y ML, como Pandas, Matplotlib, Scikit-learn, facilitando un flujo de trabajo cohesivo y eficiente (VanderPlas, 2016).

El preprocesamiento de datos es un paso crítico en cualquier flujo de trabajo de análisis de datos o machine learning. NumPy facilita este proceso a través de diversas funciones que permiten manipular datos, incluyendo:

- Normalización: Convertir los datos a una escala común sin distorsionar las diferencias en los rangos de valores.

```
import numpy as np
data = np.array([[0, 1], [2, 3], [4, 5]])
normalized_data = (data - np.min(data, axis=0)) / (np.max(data,
axis=0) - np.min(data, axis=0))
```

- Manejo de Datos Faltantes: Identificar y tratar con datos faltantes para evitar errores en el análisis.

```
data = np.array([[1, np.nan, 3], [4, 5, np.nan]])
clean_data = np.nan_to_num(data, nan=-999)
```

- Redimensionamiento de Arrays: Cambiar la forma de los datos para cumplir con los requisitos de entrada de algoritmos específicos.

```
data = np.arange(6)
reshaped_data = data.reshape((2, 3))
```

NumPy es fundamental en la implementación de algoritmos de machine learning, desde la fase de desarrollo hasta la ejecución eficiente de modelos entrenados. Algunos ejemplos incluyen:

- Regresión Lineal: NumPy puede utilizarse para calcular los coeficientes de una regresión lineal mediante la solución de un sistema de ecuaciones lineales o mediante la técnica de mínimos cuadrados.

```
X = np.array([[1, 1], [1, 2], [2, 2], [2, 3]])
y = np.dot(X, np.array([1, 2])) + 3
```

```
coef = np.linalg.lstsq(X, y, rcond=None)[0]
```

- Clustering con K-Means: Aunque bibliotecas como Scikit-learn ofrecen implementaciones de alto nivel de K-Means, NumPy permite comprender y personalizar el algoritmo a nivel más bajo, optimizando el rendimiento para casos de uso específicos.

- Redes Neuronales: Aunque existen frameworks específicos para redes neuronales, como TensorFlow o PyTorch, NumPy se utiliza a menudo para prototipar algoritmos simples de redes neuronales, ofreciendo un entorno de bajo nivel para entender la mecánica detrás de la propagación hacia adelante y el algoritmo de retropropagación.

NumPy es una herramienta imprescindible en el arsenal de cualquier científico de datos o investigador de machine learning. Su capacidad para realizar operaciones complejas de manera eficiente lo convierte en la base sobre la cual se construyen análisis de datos y modelos de aprendizaje automático. A medida que los datos continúan creciendo en tamaño y complejidad, la importancia de NumPy en el análisis y modelado de estos datos solo se incrementará.

2.6. USO DE LA BIBLIOTECA NUMPY

2.6.1. Instalación de NumPy

Antes de comenzar, asegúrate de tener NumPy instalado. Si aún no lo tienes, puedes instalarlo utilizando pip:

```
pip install numpy
```

2.6.2 Importación de NumPy

Para empezar a usar NumPy, primero debes importarlo en tu script de Python:

```
import numpy as np
```

2.6.3. Creación de Arrays

NumPy permite crear arrays de una dimensión (vectores), dos dimensiones (matrices) y n dimensiones (Documentación oficial de NumPy., 2024).

- Array de una dimensión:

```
arr_1d = np.array([1, 2, 3, 4, 5])
print(arr_1d)
```

- Array de dos dimensiones:

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
print(arr_2d)
```

- Array de tres dimensiones:

```
arr_3d = np.array([[[1, 2], [3, 4]], [[5, 6], [7, 8]]])
print(arr_3d)
```

2.6.4. Funciones especiales para crear Arrays

- Arrays uniformemente espaciados: La función np.arange crea arrays con rangos de valores. Es similar a la función range de Python pero retorna un array de NumPy (NumPy: creación y manipulación de datos numéricos., 2024).

```
# Crea un array de 0 a 9
arr1 = np.arange(10)
print(arr1)

# Crea un array de 0 (inclusive) a 20 (exclusive) con saltos
de 2
arr2 = np.arange(0,20, 2)
print(arr2)
```

- Arrays por numero de puntos: np.linspace genera un array de valores espaciados uniformemente entre un valor de inicio y uno de fin. Es especialmente útil para crear muestras de puntos para gráficos.

```
# Crea un array de 10 valores entre 0 y 1
linear_spaced = np.linspace(0, 1, 10)
print(linear_spaced)
```

- Arrays de unos y ceros: np.ones crea un array lleno de unos, mientras que np.zeros crea un array lleno de ceros. Ambas funciones son útiles para la inicialización de arrays.

```
# Crea un array 3x3 de unos
ones = np.ones((3, 3))
print(ones)

# Crea un array 2x4 de ceros
zeros = np.zeros((2, 4))
print(zeros)
```

- Matriz identidad: np.eye genera una matriz identidad de tamaño N x N. Una matriz identidad es una matriz cuadrada con unos en la diagonal y ceros en el resto.

```
# Crea una matriz identidad 4x4
identity_matrix = np.eye(4)
print(identity_matrix)
```

- Matriz diagonal: np.diag se utiliza para crear una matriz diagonal (una matriz con valores no nulos solo en la diagonal) o para extraer la diagonal de una matriz existente.

```
# Crea una matriz diagonal a partir de un array
diagonal = np.diag([1, 2, 3, 4])
print(diagonal)
```

- Array con números aleatorios: El módulo np.random genera arrays de números aleatorios. Puede crear arrays con muestras aleatorias de una distribución uniforme, normal, o cualquier otra distribución disponible en NumPy.

```
# Crea un array 3x3 con muestras aleatorias de una
distribución uniforme [0, 1)
random_array = np.random.random((3, 3))
print(random_array)
```

2.6.5. Indexación Avanzada y Slicing.

La indexación y el slicing son técnicas poderosas en NumPy que permiten acceder y modificar partes específicas de los arrays.

- Acceso a Elementos y Slicing:

```
# Acceder al primer elemento de a
print(a[0])

# Slicing: obtener los dos primeros elementos de b
print(b[:2])

# Slicing: obtener los elementos del 3 al 5 de a
print(a[2:5])
```

```
c = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Acceder al elemento de la segunda
# fila y tercera columna
print(c[1,2])  # 6

# Acceder a la segunda y
# tercera columna de la primera fila
print(c[0,1:3]) # [2 3]

# Acceder a la primera fila
print(c[0, :])  # [1 2 3]

# Acceder a la primera columna
print(c[:, 0])  # [1 4 7]

# Acceder a la primera y segunda fila y a la
# segunda y tercera columna
print(c[:2, 1:])
```

- Indexación booleana:

```
# Crear un array de ejemplo
data = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])

# Obtener los elementos mayores que 3
print(data[data > 3])

# Obtener los elementos menores que 5
print (data[data < 5])

# Obtener los elementos entre 3 y 8
print(data[(data > 3) & (data < 8)])
```

- Indexación de matrices con arrays de índices:

```
data = np.array([10, 20, 30, 40, 50])
indices = np.array([1, 3])
# Acceder a elementos específicos usando un array de índices
print(data[indices])
```

2.6.6. Operaciones Básicas con Arrays (Broadcasting)

El broadcasting es una poderosa característica de NumPy que permite realizar operaciones aritméticas en arrays de diferentes tamaños. Por ejemplo, puedes sumar un escalar a un array o sumar arrays de diferentes dimensiones sin necesidad de duplicar datos (Documentación oficial de NumPy., 2024).

- Operaciones con escalares y arrays

```
array = np.array([1, 2, 3, 4, 5])

# Suma: Añadir un escalar
suma = array + 10
print("Suma:", suma)

# Resta: Restar un escalar
resta = array - 10
print("Resta:", resta)

# Multiplicación: Multiplicar por un escalar
multiplicacion = array * 2
print("Multiplicación:", multiplicacion)

# División: Dividir por un escalar
division = array / 2
print("División:", division)

# Potencia: Elevar a la potencia de un escalar
potencia = array ** 2
print("Potencia:", potencia)
```

- Operaciones entre Arrays:

```
a = np.array([1, 2, 3, 4, 5, 6])
b = np.array([6, 7, 8, 9, 10, 11])

# Suma elemento a elemento
suma = a + b
print(suma)


# Producto elemento a elemento
producto = a * b
print(producto)
```

2.6.7. Funciones Útiles

NumPy ofrece muchas funciones útiles para realizar para conocer las dimensiones de un array, realizar cálculos estadísticos, algebraicos, y más (Documentación oficial de NumPy., 2024).

- Dimensiones de un array

```
# Array de una dimensión
array_1d = np.array([1, 2, 3])
print("1D array:", array_1d)
print("ndim:", array_1d.ndim)  # Devuelve 1
print("shape:", array_1d.shape)  # Devuelve (3,)
print("len:", len(array_1d))  # Devuelve 3
```

```
# Array de dos dimensiones
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
print("\n2D array:", array_2d)
print("ndim:", array_2d.ndim)  # Devuelve 2
print("shape:", array_2d.shape)  # Devuelve (2, 3)
print("len:", len(array_2d))  # Devuelve 2

# Array de tres dimensiones
array_3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10,
11, 12]]])
print("\n3D array:", array_3d)
print("ndim:", array_3d.ndim)  # Devuelve 3
print("shape:", array_3d.shape)  # Devuelve (2, 2, 3)
print("len:", len(array_3d))  # Devuelve 2
```

- Reshape: Cambiar la forma de un array:

```
arr = np.arange(6)  # Crea un array de 0 a 5
arr_reshaped = arr.reshape((2, 3))  # Cambia la forma a 2x3
print(arr_reshaped)
```

- Max, Min, Suma:

```
print("Máximo:", arr.max())
print("Mínimo:", arr.min())
print("Suma:", arr.sum())
```

- Operaciones a lo largo de ejes: Para arrays multidimensionales, puedes realizar operaciones como sumas a lo largo de un eje específico:

```
print("Suma a lo largo de columnas:", arr_reshaped.sum(axis=0))
```

2.6.8. NumPy para Algebra Lineal

NumPy también es muy útil para operaciones de álgebra lineal como productos de matrices, determinantes y encontrar autovalores.

- Matriz transpuesta:

```
e=np.array([[1,2],[3,4]])

# imprime la matriz transpuesta de e
print(e.T)
```

- Producto de matrices:

```
vec1 = np.array([1, 2, 3])
vec2 = np.array([4, 5, 6])

# Producto punto
productoPunto = np.dot(vec1, vec2)
print(productoPunto)

mat1 = np.array([[1, 2], [3, 4]])
mat2 = np.array([[5, 6], [7, 8]])

# Producto de matrices
producto = np.dot(mat1, mat2)
print(producto)
```

- Producto Cruz:

```
# Definir dos vectores en 3D
v1 = np.array([1, 2, 3])
v2 = np.array([4, 5, 6])

# Calcular el producto cruz
producto_cruz = np.cross(v1, v2)
print(producto_cruz)
```

NumPy proporciona un submódulo de álgebra lineal, numpy.linalg, que permite realizar varias operaciones de álgebra lineal (Documentación oficial de NumPy., 2024).

- Sistemas de ecuaciones lineales:

```
# Sistema de ecuaciones 3x + y = 9; x + 2y = 8
a = np.array([[3, 1], [1, 2]])
b = np.array([9, 8])
x = np.linalg.solve(a, b)
print(x)
```

- Determinante de una matriz

```
d= np.array([[1,2],[3,4]])
d= np.linalg.det(d) #: Calcula el determinante de una matriz.
print (d)
```

- Inversa de una matriz

```
i=np.array([[1,2],[3,4]])
inversa= np.linalg.inv(i) #: Calcula la inversa de una matriz.
print (inversa)
```

2.6.9. Manipulación de Formas

Transformar la forma de los arrays es una operación común en el preprocesamiento de datos.

- Flattening: Convertir un array multidimensional en un vector:

```
arr_2d = np.array([[1, 2, 3], [4, 5, 6]])
arr_flattened = arr_2d.flatten()
print(arr_flattened)
```

- Transposición: Cambiar filas por columnas y viceversa:

```
arr_transposed = arr_2d.T
print(arr_transposed)
```

2.6.10. Operaciones Estadísticas

NumPy ofrece funciones para realizar operaciones estadísticas básicas, lo que es especialmente útil en el análisis de datos.

```
data = np.array([1, 2, 3, 4, 5])
print("Media:", np.mean(data))
print("Mediana:", np.median(data))
print("Desviación estándar:", np.std(data))
```

2.6.11. Trabajo con Datos en Formato Numérico

A menudo, los datos con los que trabajas pueden no estar en formato numérico o pueden contener valores no válidos (NaNs). NumPy te permite manejar estos casos de manera eficiente.

- Reemplazo de NaN por un valor fijo:

```
data = np.array([1, np.nan, 3, 4, np.nan])
data_clean = np.nan_to_num(data, nan=0)
print(data_clean)
```

- Conversión de tipos de datos:

```
data = np.array([1, 2, 3, 4, 5])
# Convertir a float
data_float = data.astype(float)
print(data_float)

data = np.array([1.1, 2.2, 3.3, 4.4, 5.5])
# Convertir a int
data_int = data.astype(int)
print(data_int)
```

2.6.12. Uso de np.meshgrid

np.meshgrid es particularmente útil para generar coordenadas de mallas, comúnmente utilizado en gráficos y simulaciones físicas.

```
x = np.array([0, 1, 2])
y = np.array([0, 1])
X, Y = np.meshgrid(x, y)
print("X:\n", X)
print("Y:\n", Y)
```

2.6.13. Funciones Universales (Ufuncs)

Las funciones universales (ufuncs) son funciones que operan en arrays de NumPy elemento por elemento, permitiendo cálculos vectorizados rápidos (Documentación oficial de NumPy., 2024).

Existen dos tipos principales de funciones universales en NumPy:

- **Ufuncs unarias:** Operan sobre un solo input. Ejemplos incluyen operaciones como np.sqrt (raíz cuadrada), np.exp (exponencial), np.log (logarítmica), y las funciones trigonométricas como np.sin, np.cos, etc.
- **Ufuncs binarias:** Operan sobre dos inputs. Ejemplos incluyen operaciones aritméticas como np.add, np.subtract, np.multiply, np.divide, así como operaciones lógicas como np.greater (mayor que), np.less (menor que), etc.

A continuación, se muestran algunos ejemplos sobre uso de funciones universales

```
# Crear dos arrays
a = np.array([1, 2, 3, 4])
b = np.array([5, 6, 7, 8])
```

```
# Uso de una ufunc binaria: np.add
result = np.add(a, b)  # Suma elemento por elemento
print(result)  # [ 6  8 10 12]

# Uso de una ufunc unaria: np.sqrt
sqrt_a = np.sqrt(a)
print(sqrt_a)  # [1.         1.41421356 1.73205081 2.        ]
```

- Crear tu propia ufunc:

  NumPy permite crear ufuncs personalizadas a través de np.frompyfunc, aunque para cálculos complejos se recomienda explorar otras opciones que preserven la velocidad de cálculo.

```
# Definir una función Python simple que
#operará elemento por elemento
def my_func(x, y):
    return x + y * 2

# Convertir esta función en una ufunc
# con np.frompyfunc Los argumentos son la función a convertir,
# el número de inputs y el número de outputs
my_ufunc = np.frompyfunc(my_func, 2, 1)

# Ahora, puedes usar my_ufunc sobre ndarrays de NumPy
result = my_ufunc(np.array([1, 2, 3]), np.array([4, 5, 6]))
print(result)  # [9 12 15]
```

2.3.14. Manipulación de Datos con NumPy

- Concatenación de arrays: NumPy facilita la combinación de múltiples arrays, ya sea vertical o horizontalmente, mediante np.concatenate, np.vstack y np.hstack.

```
a = np.array([[1, 2], [3, 4]])
 = np.array([[5, 6]])
print(np.vstack([a, b]))  # Apila verticalmente
```

- División de arrays: Puedes dividir arrays en múltiples sub-arrays usando np.split, np.hsplit o np.vsplit.

```
c = np.array([1, 2, 3, 4, 5, 6])
first, second, third = np.split(c, [2, 4])  # Divide c en [1,
2], [3, 4] y [5, 6]
print(first, second, third)
```

2.6.15. Trabajo con Fechas y Tiempos

NumPy también ofrece soporte para trabajar con fechas y tiempos a través de np.datetime64 y np.timedelta64.

```
today = np.datetime64('today', 'D')
tomorrow = today + np.timedelta64(1, 'D')
print("Today:", today)
print("Tomorrow:", tomorrow)
```

2.6.16. Guardar y Cargar Datos con NumPy

NumPy proporciona funciones para guardar y cargar datos desde archivos, lo que es útil para trabajar con conjuntos de datos persistentes.

- Guardar a un archivo:

```
np.save('mi_array.npy', a)  # Guarda el array a en el archivo
                            # mi_array.npy
```

- Cargar desde un archivo:

```
a_cargado = np.load('mi_array.npy')
print(a_cargado)
```

CAPUTILO 3. MATPLOTLIB

Matplotlib es una de las bibliotecas más populares y utilizadas para la creación de gráficos estáticos, animados e interactivos en Python. Te permitirá visualizar tus datos de manera clara y efectiva.

3.1. INSTALACIÓN DE MATPLOTLIB

Si aún no tienes Matplotlib instalado, puedes hacerlo fácilmente utilizando pip:

```
pip install matplotlib
```

3.2. IMPORTACIÓN DE MATPLOTLIB

Para comenzar a usar Matplotlib en tu código, primero debes importar matplotlib.pyplot, que es el módulo que contiene la mayoría de las funciones de graficación. Por convención, se importa como plt:

```
import matplotlib.pyplot as plt
```

3.3. CREACIÓN DE UN GRÁFICO SIMPLE

Un gráfico simple de líneas se puede crear rápidamente con unos pocos datos (Documentación oficial de Matplotlib., 2024).

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

# Datos
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Crear un gráfico de líneas
plt.plot(x, y)
plt.show()
```

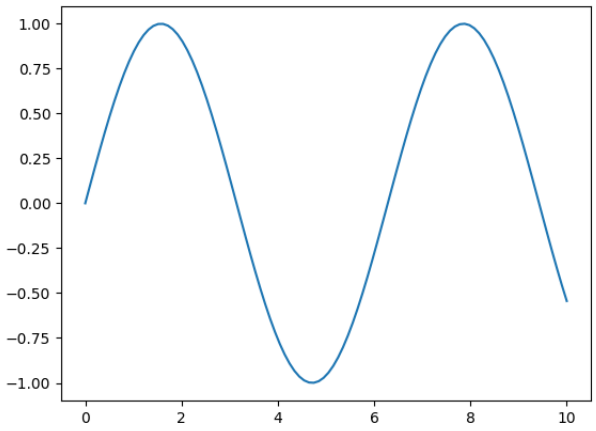


Figura 2. Gráfica de la función senoen matplotlib

3.4. PERSONALIZACIÓN BÁSICA DEL GRÁFICO

Matplotlib ofrece muchas opciones para personalizar tus gráficos, desde el color y el estilo de las líneas hasta las etiquetas de los ejes.

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

# Datos
x = np.linspace(0, 10, 100)
y = np.sin(x)

# Personaliza color, estilo de línea y grosor
plt.plot(x, y, color='blue', linestyle='--', linewidth=2)
plt.xlabel('Eje X')  # Etiqueta del eje X
plt.ylabel('Eje Y')  # Etiqueta del eje Y
plt.title('Gráfico de Seno')  # Título del gráfico
plt.show()
```

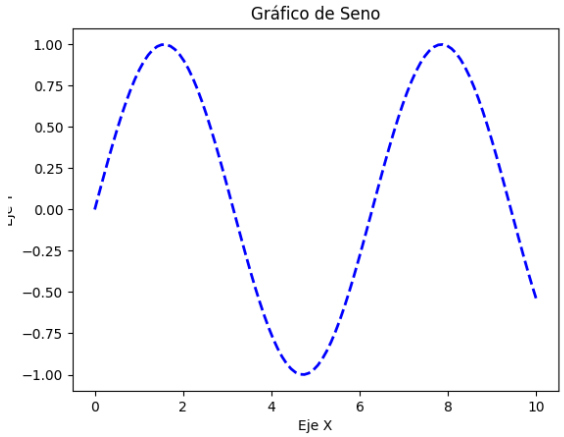


Figura 3. Grafica con estilo personalizado

3.5. TIPOS DE GRÁFICOS

Matplotlib soporta una amplia variedad de gráficos, incluyendo histogramas, gráficos de dispersión, gráficos de barras, y más.

3.5.1. Gráfico de Dispersión

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


x = np.random.rand(50)
y = np.random.rand(50)

plt.scatter(x, y)
plt.title('Gráfico de Dispersión')
plt.show()
```

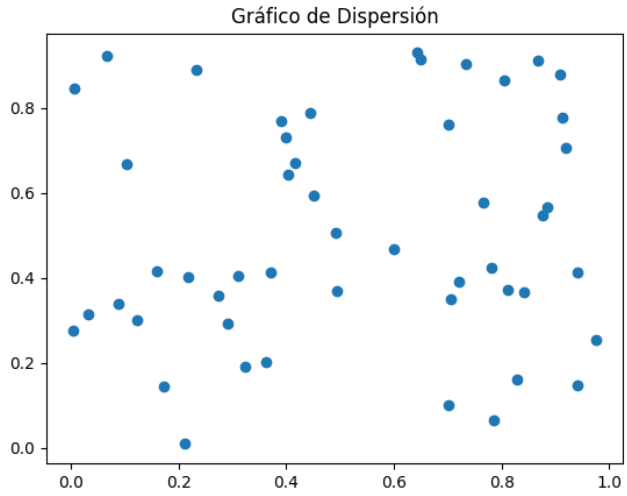


Figura 4. Ejemplo de gráfico de dispersión

3.5.2. Histograma

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


data = np.random.randn(1000)

plt.hist(data, bins=30, alpha=0.5, color='steelblue')
plt.title('Histograma')
plt.show()
```

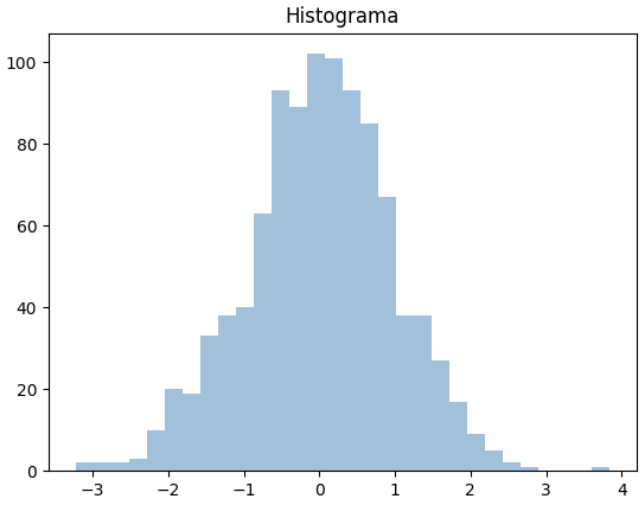


Figura 5. Ejemplo de histograma

3.5.3. Gráfico de Barras

```
import matplotlib.pyplot as plt
```

```
import numpy as np

categorias = ['A', 'B', 'C', 'D']
valores = [10, 20, 15, 5]

plt.bar(categorias, valores)
plt.title('Gráfico de Barras')
plt.show()
```

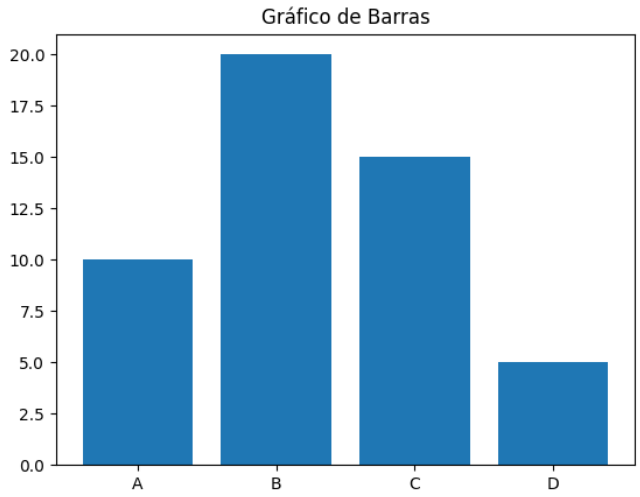


Figura 6. Gráfico de barras

3.5.4. Múltiples Gráficos

Matplotlib facilita la creación de múltiples gráficos en una sola figura, organizándolos en una cuadrícula (Matplotlib: Gráficas usando pylab., 2024).

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

x = np.random.rand(50)
y = np.random.rand(50)
categorias = ['A', 'B', 'C', 'D']
valores = [10, 20, 15, 5]
data = np.random.randn(1000)


fig, ax = plt.subplots(2, 2)  # Crea una cuadrícula de 2x2 para los
gráficos

ax[0, 0].plot(x, y, color='green')
ax[0, 1].scatter(x, y, color='red')
ax[1, 0].bar(categorias, valores, color='blue')
ax[1, 1].hist(data, bins=30, alpha=0.5, color='orange')

plt.show()
```

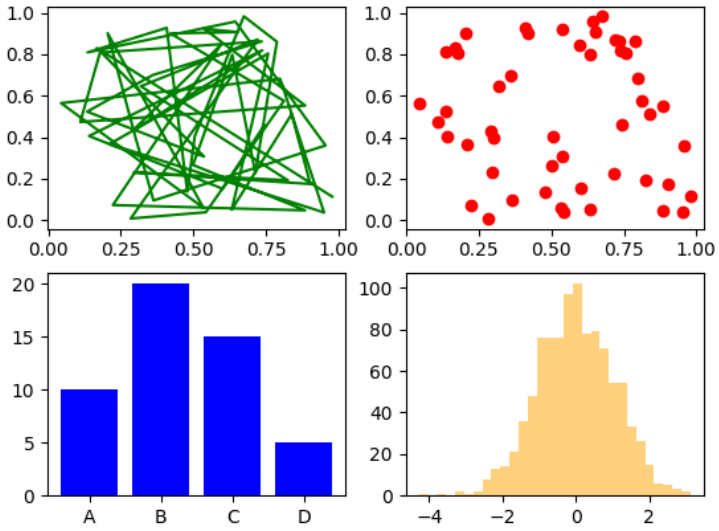


Figura 7. Múltiples gráficos en Matplotlib

3.6. GUARDANDO GRÁFICOS

Puedes guardar fácilmente tus gráficos en una variedad de formatos de archivo, como PNG, PDF, SVG, entre otros.

```
plt.plot(x, y)
plt.savefig('mi_grafico.png')
```

Para publicaciones o presentaciones, es posible que necesites guardar tus gráficos en alta resolución.

```
plt.plot(x, np.sin(x))
# Guarda el gráfico en alta resolución
plt.savefig('sinx.png', dpi=300)
```

3.7. SUBPLOTS

Aunque ya hemos visto cómo crear múltiples gráficos en una sola figura, Matplotlib ofrece aún más control sobre la disposición de los subplots, lo que te permite crear diseños complejos y personalizados.

3.7.1. Ajuste de Espaciado

Puedes ajustar el espaciado entre los subplots para asegurarte de que las etiquetas, títulos y gráficos no se solapen (Documentación oficial de Matplotlib., 2024).

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

x = np.random.rand(50)
y = np.random.rand(50)
categorias = ['A', 'B', 'C', 'D']
valores = [10, 20, 15, 5]
data = np.random.randn(1000)

fig, ax = plt.subplots(2, 2, figsize=(10, 10))
fig.subplots_adjust(hspace=0.4, wspace=0.4)  # Ajusta el espaciado
                                             # horizontal y vertical

# Llena cada subplot con datos
ax[0, 0].plot(x, y)
ax[0, 0].set_title('Gráfico Superior Izquierdo')

ax[0, 1].scatter(x, y)
ax[0, 1].set_title('Gráfico Superior Derecho')

ax[1, 0].bar(categorias, valores)
ax[1, 0].set_title('Gráfico Inferior Izquierdo')

ax[1, 1].hist(data, bins=30)
ax[1, 1].set_title('Histograma Inferior Derecho')

plt.show()
```

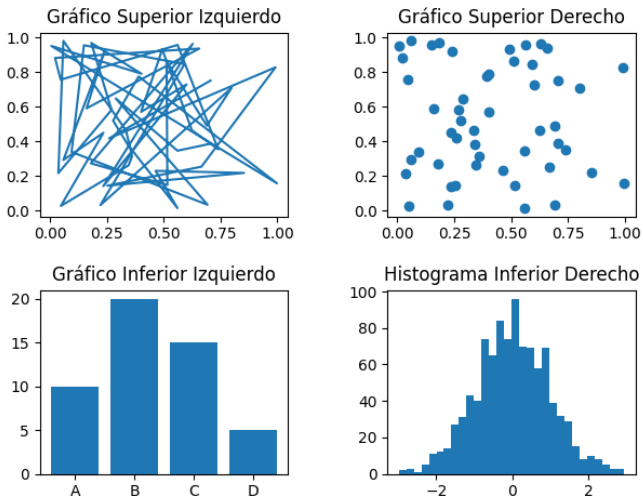


Figura 8. Ajuste de espacios en subplots

3.7.2. Gráficos de Ejes Compartidos

Para comparar diferentes conjuntos de datos que comparten ejes comunes, puedes vincular los ejes de varios subplots. Esto es especialmente útil cuando las escalas son idénticas o cuando deseas destacar diferencias o similitudes (Matplotlib: Gráficas usando pylab., 2024).

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

x = np.linspace(0, 10, 100)

fig, ax = plt.subplots(2, sharex='col', sharey='row')
ax[0].plot(x, np.sin(x))
ax[1].plot(x, np.cos(x))
plt.show()
```

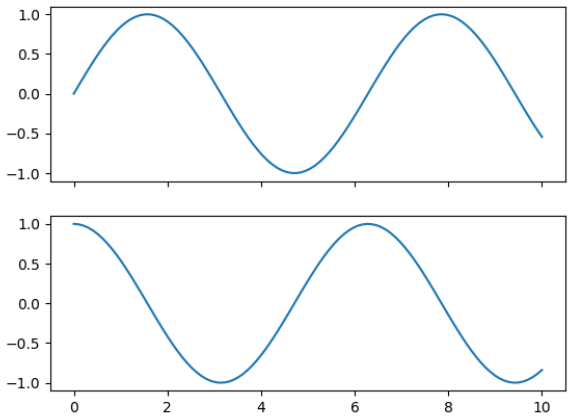


Figura 9. Gráficos de Ejes Compartidos

3.8. GRÁFICOS CON DOBLE EJE Y

Crear un gráfico que utilice dos escalas diferentes para el eje Y es útil cuando tienes variables con diferentes magnitudes pero quieres visualizarlas juntas (Documentación oficial de Matplotlib., 2024).

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

x = np.linspace(0, 10, 100)

fig, ax1 = plt.subplots()

color = 'tab:red'
ax1.set_xlabel('x')
ax1.set_ylabel('sin(x)', color=color)
ax1.plot(x, np.sin(x), color=color)
ax1.tick_params(axis='y', labelcolor=color)
```

```
ax2 = ax1.twinx()  # Instancia un segundo eje
                   # que comparte el mismo eje X
color = 'tab:blue'
ax2.set_ylabel('cos(x)', color=color)
ax2.plot(x, np.cos(x), color=color)
ax2.tick_params(axis='y', labelcolor=color)

fig.tight_layout()
plt.show()
```

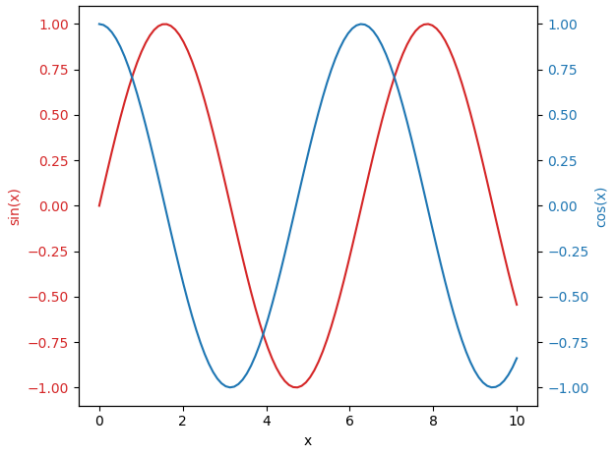


Figura 10. Gráficos don doble eje Y

3.9. PERSONALIZACIÓN Y ESTILOS

Matplotlib permite una personalización profunda de casi todos los aspectos de tus gráficos, desde colores y estilos de línea hasta fuentes y leyendas.

3.9.1. Personalización de Leyendas

Las leyendas son esenciales para entender los datos presentados. Puedes personalizar su apariencia y posición (Matplotlib: Gráficas usando pylab., 2024).

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

x = np.linspace(0, 10, 100)


plt.plot(x, np.sin(x), label='sin(x)')
plt.plot(x, np.cos(x), label='cos(x)')
# Ajusta la posición y el marco
plt.legend(loc='upper right', frameon=False)
plt.show()
```

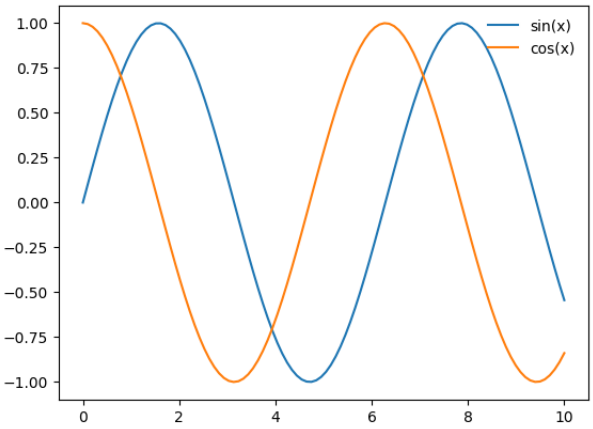


Figura 11. Personalización de Leyendas

3.9.2. Uso de Estilos Predefinidos

Matplotlib viene con una serie de estilos predefinidos que te permiten cambiar rápidamente la apariencia de tus gráficos (Documentación oficial de Matplotlib., 2024).

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

x = np.linspace(0, 10, 100)



plt.style.use('ggplot')
plt.plot(x, np.sin(x))
plt.show()

plt.style.use('seaborn')
plt.plot(x, np.cos(x))
plt.show()
```

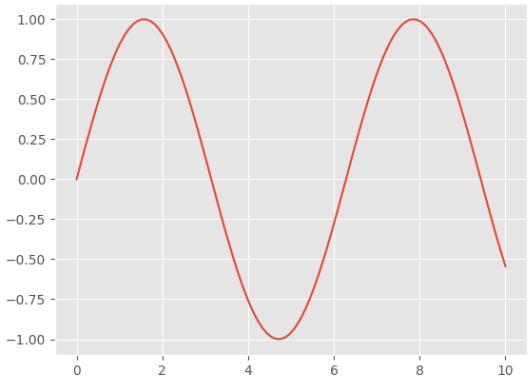


Figura 12. Uso de Estilos Predefinidos (ejemplo 1)

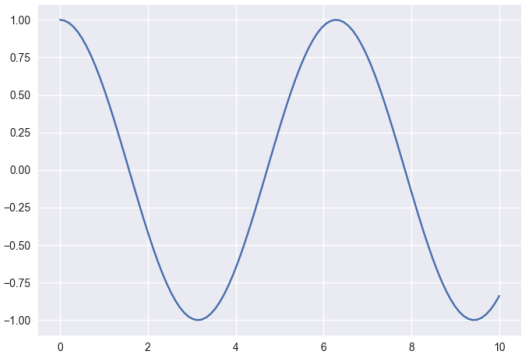


Figura 13. Uso de Estilos Predefinidos (Ejemplo 2)

3.10. GRÁFICOS DE ÁREA

Los gráficos de área son útiles para comparar cantidades a lo largo del tiempo o entre categorías distintas, con el área bajo la curva rellena (Documentación oficial de Matplotlib., 2024).

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

x = np.linspace(0, 10, 100)


x = np.arange(1, 6)
y1 = np.array([1, 4, 6, 8, 4])
y2 = np.array([2, 2, 7, 10, 5])

plt.fill_between(x, y1, color="skyblue", alpha=0.4)
```

```
plt.fill_between(x, y2, color="olive", alpha=0.4)
plt.plot(x, y1, color="Slateblue", alpha=0.6, linewidth=2)
plt.plot(x, y2, color="olive", alpha=0.6, linewidth=2)
plt.show()
```

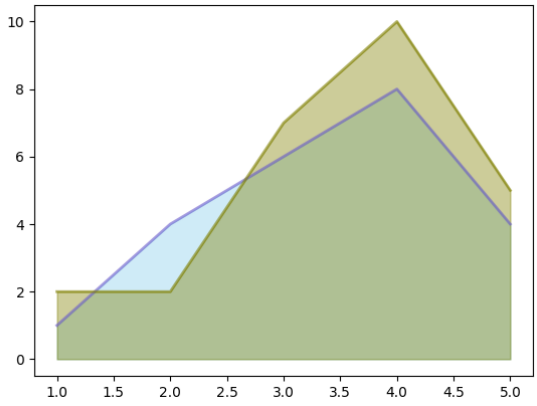


Figura 14. Gráficos de área

3.11. GRÁFICOS DE BARRAS APILADAS

Los gráficos de barras apiladas permiten ver la distribución de varias categorías a lo largo del tiempo o en diferentes condiciones, apilando los datos en barras verticales (Documentación oficial de Matplotlib., 2024).

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

x = np.linspace(0, 10, 100)


N = 5
menMeans = (20, 35, 30, 35, 27)
womenMeans = (25, 32, 34, 20, 25)
ind = np.arange(N)

plt.bar(ind, menMeans, width=0.35, label='Hombres')
plt.bar(ind, womenMeans, width=0.35, bottom=menMeans,
label='Mujeres')

plt.ylabel('Puntuaciones')
plt.title('Puntuaciones por grupo y genero')
plt.xticks(ind, ('G1', 'G2', 'G3', 'G4', 'G5'))
plt.yticks(np.arange(0, 81, 10))
plt.legend(loc='best')
plt.show()
```

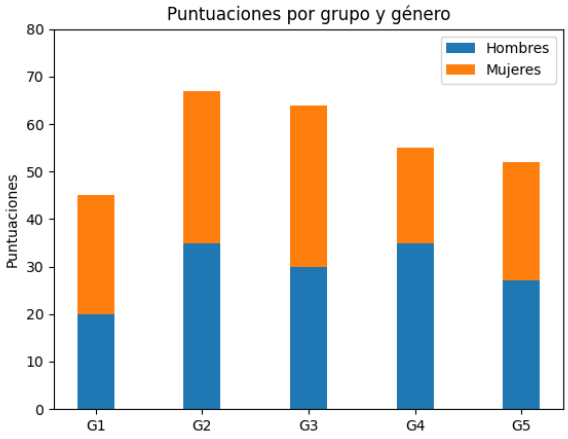


Figura 15. Gráfico de barras apiladas

3.12. GRÁFICOS DE CAJA (BOX PLOTS)

Los box plots son una forma estandarizada de mostrar la distribución de datos basada en un resumen de cinco números (“mínimo”, primer cuartil (Q1), mediana, tercer cuartil (Q3) y “máximo”), ayudando a identificar outliers y comprender la dispersión de los datos (Documentación oficial de Matplotlib., 2024).

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

data = np.random.rand(10, 5)

plt.boxplot(data)
plt.title('Box plot')
plt.show()
```

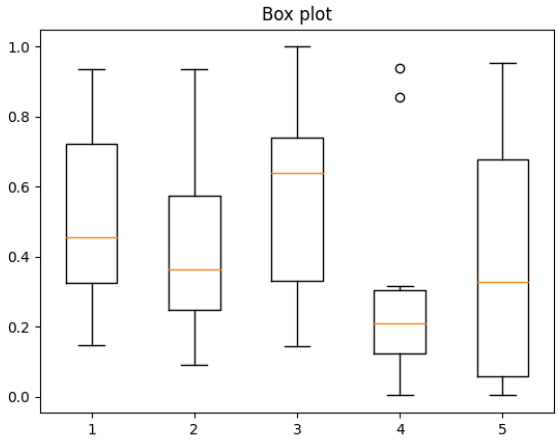


Figura 16. Gráficos de caja

3.13. GRÁFICOS DE VIOLÍN

Los gráficos de violín combinan las propiedades de los box plots y los KDE (Kernel Density Estimation), proporcionando una representación más rica de la distribución de los datos (Documentación oficial de Matplotlib., 2024).

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

data = np.random.rand(10, 5)

plt.violinplot(data)
plt.title('Violin plot')
plt.show()
```

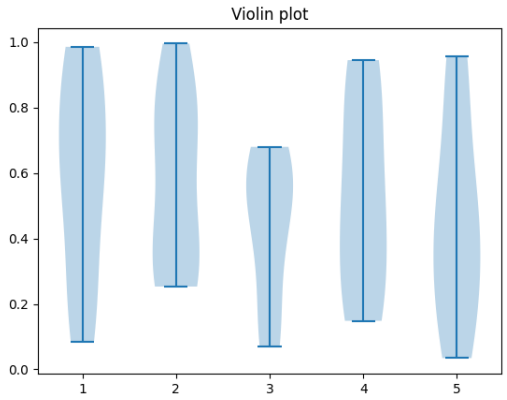


Figura 17. Gráficos de violín

3.14. MAPAS DE CALOR (HEATMAPS)

Los mapas de calor son útiles para visualizar la magnitud de los fenómenos representados por colores, siendo comúnmente usados para representar la correlación entre variables (Matplotlib: Gráficas usando pylab., 2024).

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

matrix_data = np.random.rand(10,10)

plt.imshow(matrix_data, cmap='hot', interpolation='nearest')
plt.title('Heatmap Example')
plt.colorbar()
plt.show()
```

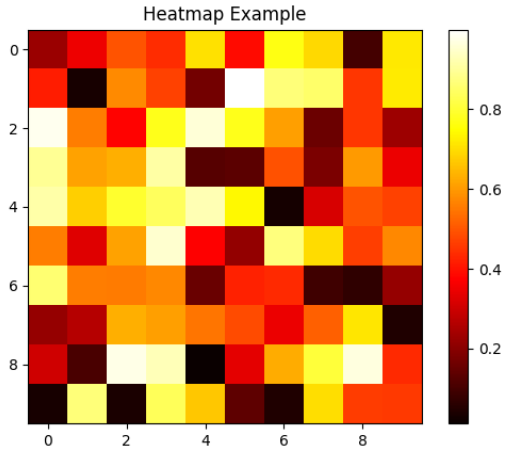


Figura 18. Gráfico de mapa de calor

3.15. GRÁFICOS DE CONTORNO

Los gráficos de contorno son útiles para representar funciones tridimensionales en dos dimensiones, donde las líneas de contorno indican áreas de igual altura (Documentación oficial de Matplotlib., 2024).

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

x = np.linspace(-3.0, 3.0, 100)
y = np.linspace(-3.0, 3.0, 100)
X, Y = np.meshgrid(x, y)
Z = np.sqrt(X**2 + Y**2)

plt.contour(X, Y, Z)
plt.title('Contour Plot')
plt.xlabel('x')
plt.ylabel('y')
plt.colorbar()
plt.show()
```

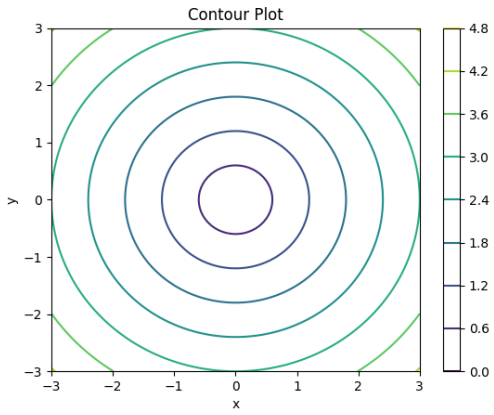


Figura 19. Gráficos de contorno

3.16. POLAR CHARTS

Los gráficos polares son útiles para visualizar datos donde las mediciones están distribuidas en ángulos desde un punto central (Documentación oficial de Matplotlib., 2024).

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

theta = np.linspace(0, 2*np.pi, 100)
r = np.abs(np.sin(5*theta))

plt.polar(theta, r)
plt.title('Polar Chart')
plt.show()
```

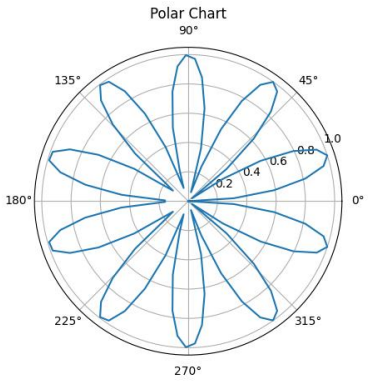


Figura 20. Gráficos polares

3.17. GRÁFICOS 3D

Matplotlib también soporta gráficos 3D, lo que te permite visualizar datos tridimensionales. Para usar gráficos 3D, necesitas importar el módulo `Axes3D` de `mpl_toolkits.mplot3d` (Documentación oficial de Matplotlib., 2024).

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

x = np.linspace(-5, 5, 100)
y = np.linspace(-5, 5, 100)
X, Y = np.meshgrid(x, y)
Z = np.sin(np.sqrt(X**2 + Y**2))

ax.plot_surface(X, Y, Z, cmap='viridis')

plt.title('Surface Plot 3D')
plt.show()
```

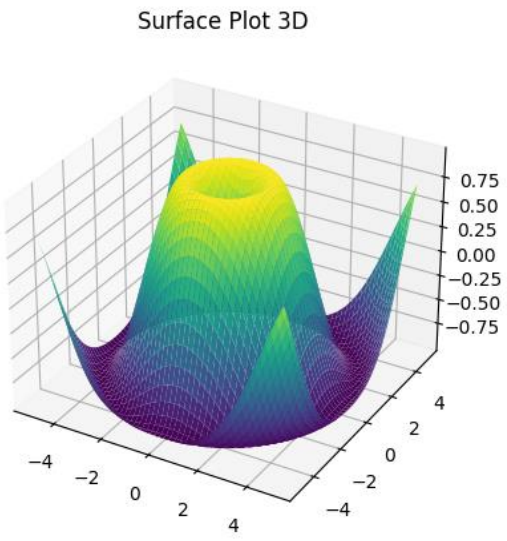


Figura 21. Gráficos en 3D

CONCLUSIONES

- NumPy es esencial para el análisis de datos en Python, proporcionando una base sólida para la manipulación eficiente de datos a través de su estructura de array n-dimensional.

- Matplotlib es una herramienta versátil para la visualización de datos, permitiendo la creación de una amplia gama de gráficos estáticos, animados e interactivos para interpretar datos complejos.

- La integración de NumPy con Matplotlib facilita el análisis y la visualización de datos, donde NumPy maneja el procesamiento y Matplotlib se encarga de la presentación visual.

- Las operaciones vectorizadas de NumPy mejoran significativamente el rendimiento, al permitir operaciones matemáticas y lógicas sobre arrays completos sin necesidad de bucles explícitos.

- NumPy y Matplotlib son fundamentales en la ciencia de datos y el aprendizaje automático, siendo herramientas clave en la carga, manipulación, análisis y visualización de grandes conjuntos de datos.

ANEXO A. BUENAS PRÁCTICAS DE PROGRAMACIÓN

La programación es una actividad creativa y altamente técnica que presenta muchos desafíos. Como todo proceso de construcción, requiere de una serie de principios rectores que garanticen resultados óptimos, eficientes y confiables. En este apéndice exploraremos algunas de las mejores prácticas establecidas por expertos para lograr un código de calidad.

Si bien no existen reglas absolutas, adoptar las siguientes recomendaciones en la medida de lo posible puede marcar una gran diferencia en la facilidad de desarrollo, mantenimiento y confiabilidad nuestros programas (McConnell, 2004) (Hunt, 1999). Después de todo, ¡la única regla inquebrantable es que funcionen!

Formateo consistente de código

Un código con formato limpio, ordenado y estandarizado facilita su lectura y comprensión significativamente (Stellman, 2020). Esto aplica tanto para nosotros mismos como para otros programadores que deban leerlo. Un estilo de codificación coherente nos ayuda a reconocer patrones rápidamente.

Algunas directrices para lograr esto:

• Ser consistentes en el uso de sangrías e indentaciones para delimitar bloques y métodos. La mayoría de los lenguajes como Python usan 4 espacios por tabulación.

• Limitar las líneas de código a una longitud máxima recomendada de 79 caracteres para evitar desplazamiento horizontal (Wilson Giese, 2019).

• Separar bloques de código relacionados verticalmente con líneas en blanco.

• Incluir comentarios relevantes y omitir los innecesarios.

Un buen formateo visual reduce la carga cognitiva para comprender qué está ocurriendo en el código. Esto es crítico en secciones complejas. Siempre podemos complementar con comentarios para explicar la lógica detrás.

Programación modular

La modularidad consiste en dividir un programa en partes más pequeñas, independientes y manejables, conocidas como módulos o componentes. Cada módulo se especializa en una tarea específica. Esta técnica presenta grandes beneficios (McLaughlin, 2006):

• Reutilización: Los módulos pueden emplearse fácilmente en otros proyectos.

• Organización: Permite focalizarse en porciones pequeñas aisladas.

• Mantenimiento: Es más sencillo realizar cambios en una pieza independiente.

En Python es común crear módulos separando funcionalidades en diferentes archivos o carpetas. También se aplican conceptos de Programación Orientada a Objetos mediante clases. Idealmente cada clase o módulo debe ser autocontenido y realizar una única función de negocio, la modularidad promueve código más flexible y gestionable.

Simplicidad y legibilidad

La simplicidad en la programación es una meta que todos perseguimos. Un código simple tiende a minimizar la complejidad innecesaria lo máximo posible (Martin, 2008). Esto hace que sea más fácil de leer y entender para humanos, una característica invaluable.

Podemos pensar en la simplicidad en varias dimensiones:

- Minimizar duplicación mediante funciones y módulos reutilizables.
- Evitar código denso y difícil de comprender. Siempre que sea posible, es preferible dividir en pedazos más pequeños.
- Limitar anidamientos profundos de estructuras de control como bucles y condicionales.
- Descomponer problemas en trozos manejables directos en lugar de soluciones complejas.

La legibilidad se relaciona con qué sencillo es para otra persona entender nuestro código (Gorman, 2015). Un código ilegible genera una gran fricción cada vez que necesitamos modificarlo o corregir errores.

Algunas sugerencias:

- Usar variables y nombres significativos que den pistas sobre su utilidad y contenido.
- Preferir nombres largos y completos vs. abreviaturas crípticas.
- Añadir comentarios relevantes para complementar el código.

La simplificación y legibilidad pueden conseguirse gradualmente refactorizando código existente o prestando especial atención en los nuevos desarrollos.

Documentación efectiva

La documentación efectiva es esencial para entender un programa a profundidad, tanto para nosotros en el futuro como para otros potenciales colaboradores (Krile, 2006). Incluir artefactos explicativos relevantes es una cortesía que ahorra muchas horas de investigación.

Tipo de documentos útiles:

- Comentarios en el código mismo para partes clave o complejas.
- Documentación técnica con diagramas de clases y módulos.
- Documentación de la API para usuarios finales (ReadMe, docstrings).
- Ejemplos de código y tutoriales paso a paso (Microsoft, Comment your code, 2022a).

La documentación más valiosa es la que se mantiene actualizada. Una estrategia es crearla junto al código durante el desarrollo. También debemos evitar sobre-documentar y preferir calidad sobre cantidad (Brown, 2011). Comentarios irrelevantes contribuyen al ruido.

ANEXO B. REGLAS DE NOMENCLATURA PARA NOMBRE DE IDENTIFICADORES

A la hora de programar, uno de los retos frecuentes es decidir qué nombres asignar a los diferentes identificadores que utilizamos, como variables, funciones, clases, módulos y otros elementos. Definir una estrategia clara y consistente de nomenclaturas tiene gran impacto en la legibilidad, mantenimiento y calidad de nuestro código.

En este anexo exploramos algunas convenciones recomendadas por la comunidad para escoger los nombres de identificadores de forma óptima en diversos lenguajes de programación.

Longitud de los identificadores

Un primer debate recurrente es si utilizar nombres largos o cortos para nuestras variables y componentes (AurumByte, 2017) (Utsav, 2021). Existen diferencias marcadas según lenguajes:

- En Python, Ruby y Go prefieren nombres completos no abreviados para una máxima claridad.
- C# y Java también favorecen nombres explícitos pero suelen ser más concisos.
- En C y C++ históricamente usaban identificadores muy cortos para ahorrar digitación.

En términos generales, se recomienda encontrar un balance entre concisión y significado pleno. Los límites de longitud presentan un tope natural para forzar a condensar más.

Convenciones según tipo de identificador

Las convenciones de nomenclatura también pueden variar según la categoría:

- Variables y funciones: nombres en minúscula separados por guiones bajos. Ej: dias_de_pago, obtener_totales().
- Constantes: mayúsculas separadas también por guiones. Ej: MAX_USERS, COLOR_ROJO.

- Clases: primer letra mayúscula de cada palabra (UpperCamelCase). Ej: FacturaDeVenta, ClientePreferencial.
- Módulos: igual que clases pero sin espacios o guiones. Ej: PrincipalAnalytics, RoboticVisionModule.
- Métodos y propiedades: primer verbo/sustantivo en minúscula luego UpperCamelCase.
- Ej: obtener_saldo(), nombreCompleto. (Gómez, 2020) (Microsoft, Naming Guidelines, 2020b)

Nombres significativos

Más allá de las convenciones sintácticas, es crucial que los nombres aporten valor semántico representando los datos o conducta del identificador lo más precisamente posible dentro del dominio de la aplicación.

Unas recomendaciones para lograrlo son:

- Usar nombres completos en lugar de abreviaciones crípticas.
- Respetar consistentemente palabras y conceptos específicos del dominio.
- Para variables booleanas que indican estado, prefijar con “is”, “has”, “can", etc. Ej: isAuthorized, hasPermisosAdmin.
- Incluir unidades, tipos y propósito en los nombres de variables numéricas.
- Preferir nombres activos de acciones para métodos (get, calculate, print). (Python, 2020). (Microsoft, Naming Guidelines, 2020b)

En resumen, definir reglas uniformes de nomenclatura como las presentadas facilita la lectura fluida del código y reduce la carga cognitiva asociada para cualquiera que deba comprenderlo. Forma parte de las mejores prácticas de un programador profesional.

REFERENCIAS

- AurumByte. (2017). *Naming Conventions in Programming Languages*. Obtenido de https://www.aurumbyte.com/naming-convention-types-in-programming-languages/
- Brown, W. J. (2011). *Does Code Documentation Save Money? A Controlled Experiment with Professional Developers. Center for Advanced Studies in Software Engineering (CASE). .*
- *Documentación oficial de Matplotlib.* (2024). Obtenido de https://matplotlib.org/stable
- *Documentación oficial de NumPy.* (2024). Obtenido de https://numpy.org/doc/stable
- Downey, A. (2015). En *Think Python: How to think like a computer scientist (2nd ed.).* O'Reilly Media.
- Downey, A. (2016). En *The importance of Python programming skills. In A. Downey (Ed.), Think Python.* O'Reilly Media.
- Gómez, M. J. (2020). *Python naming conventions: rules and good practices*. Obtenido de https://peps.python.org/pep-0008/#function-and-variable-names
- Gorman, M. M. (2015). *Python by Example.* Packt Publishing Ltd.
- Hunt, A. T. (1999). En *The Pragmatic Programmer(1999).* Addison-Wesley.
- Krile, T. (2006). *The Craft of Software Documentation Writing in a Lean Development Group.* . Agile Conference.
- Lott, M. (2022). En *Object-oriented programming in Python. In Beginning Python* (págs. 337-337). John Wiley & Sons.
- Lutz, M. (2013). En *Learning Python (5th ed.).* O'Reilly Media.
- *Matplotlib: Gráficas usando pylab.* (2024). Obtenido de https://claudiovz.github.io/scipy-lecture-notes-ES/intro/matplotlib/matplotlib.html
- McConnell, S. (2004). En C. complete. Pearson Education.
- McLaughlin, B. D. (2006). *Head First Object-Oriented Analysis and Design.* O'Reilly Media.
- Microsoft. (2020b). *Naming Guidelines*. Obtenido de https://learn.microsoft.com/en-us/dotnet/standard/design-guidelines/naming-guidelines
- Microsoft. (2022a). *Comment your code*. Obtenido de https://docs.microsoft.com/en-us/learn/modules/python-comments-documenting-code/

- *NumPy: creación y manipulación de datos numéricos.* (2024). Obtenido de https://claudiovz.github.io/scipy-lecture-notes-ES/intro/numpy/index.html
- Oliphant, T. E. (2006). *A guide to NumPy. USA: Trelgol Publishing. .*
- Python. (2020). *Style Guide for Python Code*. Obtenido de https://peps.python.org/pep-0008/#prescriptive-naming-conventions
- Python Software Foundation. (2022). *About Python*. Obtenido de https://www.python.org/about/
- Stellman, A. G. (2020). En *Head First Coding.* O'Reilly Media.
- Sweigart, A. (2022). En *Python basics. In Automate the Boring Stuff with Python (2nd ed.)* (págs. 1-15).
- Utsav, S. U. (2021). *Naming Convention Guide for Programmers*. Obtenido de https://towardsdatascience.com/a-naming-convention-guide-for-programmers-ec259d80df81
- VanderPlas, J. (2016). *Python Data Science Handbook: Essential Tools for Working with Data.* O'Reilly Media.
- Wilson Giese, A. M. (2019). *Clean Code Concepts in Python.* Obtenido de https://www.section.io/engineering-education/clean-code-python/

TALLERES PROPUESTOS

Taller número 1. Física Computacional en Python (Caídas Libres a Proyectiles).

Descripción del Taller: Este taller proporciona una experiencia práctica en la aplicación de principios de física utilizando la programación en Python. A través de una serie de ejercicios que van desde el cálculo de caída libre hasta el lanzamiento de proyectiles, los participantes afianzarán sus habilidades en la resolución de problemas físicos y en la programación de estructuras de control, funciones y procedimientos en Python. Finalmente, integrarán todos los ejercicios en un menú interactivo que permitirá al usuario seleccionar y ejecutar cada tarea.

Objetivos:

- Fortalecer las habilidades de programación en Python en un contexto de física.
- Aplicar conocimientos de mecánica para resolver problemas de caída libre y lanzamiento de proyectiles.
- Practicar la creación y uso de funciones y estructuras de control.
- Implementar un menú interactivo para la integración de varias subrutinas.

Metodología:

- Cada ejercicio se introducirá con una breve explicación teórica seguida de la implementación práctica.
- Los participantes trabajarán individualmente en la codificación de cada punto, con soporte constante del instructor.
- Se promoverá el uso de pares para revisión de código y discusión de soluciones.
- Cada sesión concluirá con una puesta en común de los programas desarrollados y se discutirán las diferentes soluciones y enfoques.
- Se fomentará la reflexión sobre la importancia de cada concepto físico y su representación en código.

Recursos Necesarios:

- Computadoras con Python y un editor de código instalados.
- Acceso a documentación de Python y recursos en línea para consulta.
- Materiales de referencia sobre los conceptos de física implicados.

Evaluación:

- Cada script de Python será evaluado en términos de exactitud, eficiencia y legibilidad.
- Se evaluará la capacidad de los participantes para integrar correctamente las subrutinas en el menú final.
- Se considerará la participación activa en las discusiones y revisiones de pares.

Feedback y Reflexión:

- Al final del taller, se recogerá feedback de los participantes para evaluar la comprensión y aplicación de los conocimientos adquiridos.
- Se realizará una sesión de reflexión sobre cómo la programación puede ser una herramienta poderosa para la física y otras ciencias.

Puntos del taller:

1. **Cálculo de la Caída Libre:**
   Escribe un programa en Python que calcule y muestre el tiempo que tarda un objeto en caer desde una altura determinada sin resistencia del aire. Utiliza la fórmula $t = \sqrt{\frac{2h}{g}}$, donde h es la altura en metros y g es la aceleración debido a la gravedad ($9.81 m/s^2$).
2. **Conversión de Unidades de Velocidad:**
   Crea una función que convierta la velocidad de km/h a m/s y viceversa. La función debe tomar como argumento el valor de la velocidad y el tipo de conversión.
3. **Cálculo del Desplazamiento:**
   Desarrolla un script que calcule el desplazamiento de un objeto en movimiento rectilíneo uniformemente acelerado, dados la velocidad inicial, la aceleración y el tiempo. Usa la fórmula $s = ut + \frac{1}{2}at^2$.
4. **Suma de Vectores:**
   Implementa una función en Python que tome dos vectores representados como listas (por ejemplo, $[x_1, y_1]$ y $[x_2, y_2]$) y devuelva su suma vectorial.
5. **Producto Escalar de Vectores:**
   Escribe un programa que calcule el producto escalar de dos vectores y determine el ángulo entre ellos, utilizando la fórmula del coseno del ángulo.
6. **Lanzamiento de Proyectil:**
   Crea un script para calcular el alcance máximo (R) y la altura máxima (H) alcanzada por un proyectil lanzado con una velocidad inicial (v_0) a un ángulo (θ) con la horizontal. Ignora la resistencia del aire.

Programa Integrador con Menú de Opciones: Desarrollar un programa en Python que integre todas las funcionalidades de los ejercicios anteriores en un menú interactivo, permitiendo al usuario elegir qué tarea realizar. El programa debe comenzar mostrando al

usuario un menú de opciones que incluya todas las funcionalidades de los 6 puntos anteriores y una opción 7 que permita salir del programa.

Taller número 2. Programación en Python (Subrutinas, Estructuras de Control, Arreglos y Listas).

Objetivos:

- Practicar la definición y uso de funciones y procedimientos en Python.
- Aplicar estructuras de control para resolver problemas variados.
- Manejar arreglos y listas para el procesamiento de datos.
- Integrar diversos conceptos de programación en un solo sistema.

Metodología:

- Cada ejercicio se desarrollará individualmente, seguido de una sesión de revisión grupal para discutir las soluciones y posibles mejoras.
- Se fomentará el uso de buenas prácticas de programación, incluyendo la documentación adecuada de cada función y procedimiento.

Recursos Necesarios:

- Computadora con Python instalado.
- Editor de texto o IDE para desarrollo en Python (por ejemplo, Visual Studio Code, PyCharm, Jupyter Notebook).

Evaluación:

- Cada ejercicio será revisado y evaluado basándose en la corrección, eficiencia, y legibilidad del código.
- Se realizará una demostración a través del menu para evaluar la integración efectiva de las subrutinas en un sistema funcional.

Feedback y Reflexión:

- Al final del taller, se realizará una sesión de feedback donde los participantes podrán compartir sus experiencias, dificultades encontradas y cómo las superaron.
- Se discutirán las lecciones aprendidas y la importancia de cada concepto abordado en el taller para el desarrollo de software en Python.

Puntos del taller:

1) **Calculadora Básica:** Crea una función para cada operación básica (suma, resta, multiplicación, división) que acepte dos argumentos y devuelva el resultado. Implementa una función principal que solicite al usuario números y la operación a realizar, utilizando las funciones creadas.

2) **Filtrado de Lista:** Desarrolla una función que reciba una lista de números y devuelva solo aquellos que sean pares. Utiliza un bucle y condicionales dentro de la función.

3) **Transformación de Listas con Map y Lambda:** Escribe un script que utilice map y una función lambda para convertir todas las temperaturas de una lista de grados Celsius a Fahrenheit.

4) **Sistema de Calificaciones:** Implementa una función que reciba una lista de calificaciones numéricas y devuelva una lista con las calificaciones convertidas a letras (A, B, C, D, F) según el rango en el que se encuentren.

5) **Conteo de Palabras:** Crea una función que reciba una cadena de texto y retorne un diccionario con el conteo de cuántas veces aparece cada palabra. Considera ignorar mayúsculas y signos de puntuación.

6) **Función de Búsqueda:** Desarrolla una función que busque un elemento dado dentro de una lista y devuelva su índice si lo encuentra o -1 si no está presente. No utilices métodos predefinidos como .index().

7) **Validación de Paréntesis:** Escribe una función que tome una cadena de solo paréntesis (y) y determine si la secuencia es válida (cada paréntesis abierto tiene un correspondiente paréntesis cerrado).

8) **Ordenamiento Personalizado:** Implementa una función que ordene una lista de tuplas (nombre, edad) primero por edad y luego por nombre (ambos en orden ascendente). Puedes usar la función sort o sorted con parámetros personalizados.

9) **Generador de Contraseñas:** Crea una función que genere una contraseña aleatoria que consista en una combinación de letras mayúsculas, minúsculas, números y símbolos. La longitud de la contraseña debe ser un parámetro de la función.

10) **Gestión de Agenda Telefónica:** Desarrolla un programa que utilice funciones para permitir al usuario agregar, buscar, eliminar y mostrar todos los contactos de una agenda telefónica almacenada en un diccionario.

Programa Integrador con Menú de Opciones: Desarrollar un programa en Python que integre todas las funcionalidades de los ejercicios anteriores en un menú interactivo, permitiendo al usuario elegir qué tarea realizar. El programa debe comenzar mostrando al usuario un menú de opciones que incluya las siguientes funcionalidades:

1. Operaciones Básicas (Calculadora)
2. Filtrado de Lista por Números Pares
3. Conversión de Temperaturas de Celsius a Fahrenheit
4. Sistema de Calificaciones a Letras
5. Conteo de Palabras en una Cadena
6. Búsqueda de Elemento en Lista
7. Validación de Secuencia de Paréntesis
8. Ordenamiento Personalizado de Lista de Tuplas
9. Generador de Contraseñas Aleatorias
10. Gestión de Agenda Telefónica
11. Salir del Programa

Cada opción del menú debe corresponder a una función específica implementada en los ejercicios del 1 al 10. La opción de "Salir del Programa" debe terminar la ejecución del mismo.

Taller número 3. Programación Numérica con Python y NumPy.

Descripción: Este taller está diseñado para introducir y afianzar el uso de la biblioteca NumPy en Python, con el objetivo de desarrollar habilidades en la creación y manipulación de arrays numéricos, realizar operaciones matemáticas básicas y complejas, y gestionar datos con Python para aplicaciones en ciencia de datos, ingeniería y matemáticas.

Objetivos del Taller:

- Familiarizar a los participantes con la creación y manipulación de arrays en NumPy.
- Desarrollar habilidades para realizar operaciones matemáticas básicas con arrays.
- Aplicar técnicas de indexación y slicing para el manejo efectivo de subconjuntos de datos.
- Utilizar broadcasting y funciones universales para simplificar operaciones matemáticas.
- Aprender métodos de manipulación de formas y álgebra lineal en matrices.
- Manejar y tratar datos faltantes dentro de arrays numéricos.
- Practicar el almacenamiento y carga de arrays para su uso en análisis de datos.

Metodología:

- Se combinarán breves sesiones teóricas con prácticas guiadas en computadora.
- Los ejercicios se resolverán paso a paso, con explicaciones detalladas y asistencia individualizada.
- Se promoverá la discusión grupal para explorar diferentes soluciones y enfoques.

Recursos:

- Computadoras con Python y NumPy instalados.
- Acceso a documentación en línea de NumPy y otros recursos educativos.
- Cuadernos de trabajo o espacios de codificación en línea como Jupyter Notebooks.

Evaluación:

- Se revisará la corrección de los ejercicios individuales.
- Se valorará la capacidad para integrar los conceptos aprendidos en un programa final con menú interactivo.
- Se alentará a los participantes a explicar su código y el razonamiento detrás de sus soluciones.

Contenido del Taller:

Ejercicio 1: Creación y Propiedades de Arrays

Pistas:

- Usa np.array(range(1, 11)) para crear el array.
- Para cambiar la forma, emplea el método .reshape(2, 5) en el array creado.
- Accede a las propiedades .shape, .size, y .ndim para imprimir las dimensiones, la forma y el tamaño.

Ejercicio 2: Operaciones Básicas

Pistas:

- La suma y la resta se pueden hacer directamente (a + b, a - b).
- Para el producto elemento a elemento, simplemente multiplica a y b.
- Utiliza np.sum(a) para sumar todos los elementos dentro de a.

Ejercicio 3: Indexación y Slicing

Pistas:

- Recuerda que en Python, el índice comienza en 0, así que el quinto elemento es data[4].
- Para obtener una subsección, utiliza data[2:7].

Ejercicio 4: Broadcasting y Funciones Universales

Pistas:

- Crea A usando np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]).
- Para sumar un escalar a A y aplicar una ufunc, simplemente haz A + 10 y np.sqrt(A).

Ejercicio 5: Manipulación de Formas y Algebra Lineal

Pistas:

- Usa .reshape(3, 2) para cambiar la forma de M.
- El producto punto de M y su transpuesta (M.T) se puede calcular con np.dot(M, M.T).

Ejercicio 6: Trabajo con Datos Faltantes

Pistas:

- Utiliza np.nan_to_num(data, nan=0) para reemplazar np.nan por 0.
- Calcula la media del array resultante con np.mean(data).

Ejercicio 7: Guardar y Cargar Arrays

Pistas:

- np.save('mi_array.npy', data) guarda el array data.
- Usa np.load('mi_array.npy') para cargar el array desde el archivo.

Programa Integrador con Menú de Opciones: Desarrollar un programa que incorpore todos los ejercicios anteriores en un menú interactivo, reforzando así las habilidades de programación y consolidando el conocimiento adquirido.

Taller número 4. Manejo de Arrays y Operaciones Básicas usando numpy

Descripción del Taller: Este taller está orientado a familiarizar a los estudiantes con la biblioteca NumPy de Python. A través de una serie de ejercicios prácticos, los participantes aprenderán a crear y manipular arrays unidimensionales y multidimensionales, realizar operaciones básicas y matemáticas, y aplicar técnicas de indexación y slicing. El taller está diseñado para brindar una base sólida en el manejo de estructuras de datos numéricas, fundamentales para el análisis de datos y la computación científica.

Objetivos del Taller:

- Comprender la estructura y creación de arrays unidimensionales y multidimensionales en NumPy.
- Aprender a realizar operaciones aritméticas básicas con arrays.
- Aplicar funciones matemáticas a los datos de los arrays.
- Utilizar técnicas de indexación y slicing para seleccionar datos específicos.
- Familiarizarse con la generación de datos aleatorios y las funciones de agregación en NumPy.
- Desarrollar habilidades en la transformación y redimensionamiento de arrays.

Metodología:

- Cada sesión del taller comenzará con una introducción teórica al tema del ejercicio.
- Los participantes realizarán ejercicios prácticos en sus propias computadoras con la asistencia del instructor.
- Se promoverá la interacción constante y el intercambio de ideas entre los estudiantes.
- Al final de cada ejercicio, se discutirán las soluciones y las mejores prácticas.

Recursos Necesarios:

- Computadoras con acceso a un ambiente de desarrollo de Python y NumPy instalado.
- Material de referencia en línea y documentación de NumPy.
- Proyector para demostraciones del instructor y revisión de código grupal.

Evaluación:

- Los ejercicios prácticos se revisarán para asegurar la comprensión y correcta aplicación de los conceptos.
- Se animará a los estudiantes a explicar su código y las decisiones tomadas durante la programación.
- El progreso se medirá a través de la capacidad de los estudiantes para completar los ejercicios de forma independiente.

Contenido del Taller:

1) **Crear un Array Unidimensional.**
    a. Crea un array unidimensional con los números del 1 al 10.
    b. Imprime el array.
2) **Crear un Array Multidimensional**
    a. Crea una matriz 2D con 3 filas y 3 columnas, llenada con números del 1 al 9.
    b. Imprime la matriz.
3) **Operaciones Básicas con Arrays**
    a. Crea dos arrays unidimensionales con los números del 1 al 5.
    b. Suma los dos arrays y guarda el resultado en un nuevo array.
    c. Imprime el resultado.
4) **Funciones Matemáticas**
    a. Crea un array con los números del 1 al 5.
    b. Calcula la exponencial de cada elemento del array y guarda el resultado en un nuevo array.
    c. Imprime el nuevo array.
5) **Indexación y Segmentación**
    a. Crea un array con los números del 1 al 10.
    b. Selecciona los elementos pares del array y guarda el resultado en un nuevo array.
    c. Imprime el nuevo array.
6) **Generación de Datos Aleatorios**
    a. Genera un array de 10 números aleatorios entre 0 y 1.
    b. Imprime el array.
7) **Funciones de Agregación**
    a. Crea un array con los números del 1 al 5.
    b. Calcula la media de los elementos del array.
    c. Imprime la media.
8) **Creación de Arrays con Funciones de Fábrica**
    a. Crea un array de 5 elementos, todos inicializados con el valor 7.
    b. Imprime el array.
9) **Operaciones de Alineación y Broadcasting**
    a. Crea dos arrays: uno con los números del 1 al 3 y otro con los números del 4 al 6.
    b. Suma los dos arrays utilizando broadcasting y guarda el resultado en un nuevo array.
    c. Imprime el nuevo array.
10) **Funciones de Transformación y Redimensionamiento**

a. Crea un array con los números del 1 al 6.
b. Cambia la forma del array a una matriz 2x3.
c. Imprime la matriz.

Programa Integrador con Menú de Opciones: Desarrollar un programa que incorpore todos los ejercicios anteriores en un menú interactivo, reforzando así las habilidades de programación y consolidando el conocimiento adquirido.

Taller número 5. Manipulación Avanzada de Arrays con NumPy.

Descripción del Taller: Este taller está dirigido a estudiantes y profesionales que ya tienen conocimientos básicos de la biblioteca NumPy y buscan profundizar en operaciones avanzadas de manipulación de arrays. A través de los ejercicios, los participantes aprenderán a trabajar con vectores y matrices, realizar cálculos estadísticos, y aplicar transformaciones y filtros complejos.

Objetivos del Taller:

- Crear y manipular arrays unidimensionales y bidimensionales en NumPy.
- Realizar operaciones matemáticas y estadísticas avanzadas con arrays.
- Utilizar funciones de NumPy para encontrar valores mínimos, máximos, la media y la suma de elementos en arrays.
- Aplicar indexación avanzada para modificar y extraer datos de arrays.
- Comprender y utilizar la indexación booleana para filtrar elementos en arrays.
- Ejecutar operaciones de transformación y redimensionamiento de arrays.

Metodología:

- Instrucción guiada por el facilitador seguida de ejercicios prácticos.
- Análisis de caso para cada función de NumPy utilizada.
- Trabajo en parejas y grupos pequeños para fomentar la discusión y colaboración.
- Sesiones de resolución de problemas que permiten aplicar los conceptos aprendidos.

Recursos Necesarios:

- Ordenadores con Python y la biblioteca NumPy instalada.
- Acceso a documentación oficial de NumPy y material de referencia adicional.
- Pizarra o pantalla para la demostración de ejemplos por el instructor.

Evaluación:

- Evaluación continua a través de la observación durante los ejercicios prácticos.
- Evaluación final basada en la completitud y corrección de un conjunto de ejercicios asignados.

- Retroalimentación grupal al final de cada ejercicio para reforzar los conceptos.

Contenido del Taller:

1) Cree el siguiente vector A= [2, 3,5, 1, 4 ,7 9, 8, 6, 10]
2) Cree un vector B que contenga los elementos desde el 11 hasta el 20
3) Componer un vector C formado por los vectores A y B en la misma fila respectivamente
4) encuentre el valor mínimo en el vector C haciendo uso de la función propia de Numpy
5) encuentre el valor máximo en el vector C haciendo uso de la función propia de Numpy
6) encuentre la longitud en el vector C haciendo uso de la función propia de Numpy
7) encentre el promedio de los elementos en el vector C haciendo uso de las operaciones elementales suma y división
8) Encuentre el promedio en el vector C haciendo uso de la función propia de Numpy
9) Encuentre la media en el vector C haciendo uso de la función propia de Numpy
10) Encuentre la suma en el vector C haciendo uso de la función propia de Numpy
11) Cree un vector D a partir del vector C con los elementos mayores que 5
12) Cree un vector E a partir del vector C con los elementos mayores que 5 y menores que 15
13) Cambie los elementos 5 y 15 elemento del vector C por ‘7’
14) Determine la moda del vector C
15) Ordene el Vector C de menor a mayor
16) Multiplique el vector C por 10
17) Cambie los elementos del 6 al 8 de la matriz C por 60, 70 y 80 respectivamente
18) Cambie los elementos del 14 al 16 de la matriz C por 140, 150 y 160 respectivamente

Taller número 6. Análisis Numérico y Visualización con NumPy y Matplotlib.

Descripción del Taller: Este taller combina la teoría y práctica para el manejo de arrays con NumPy y la visualización de datos con Matplotlib. Los participantes aplicarán técnicas de análisis numérico y estadístico y aprenderán a interpretar los resultados visualmente a través de gráficos y manipulación de imágenes.

Objetivos del Taller:

- Comprender y aplicar técnicas de creación y manipulación de arrays con NumPy.
- Realizar operaciones básicas y de álgebra lineal en matrices.
- Practicar el acceso y la indexación avanzada en arrays.
- Ejecutar análisis estadísticos utilizando funciones de NumPy.
- Desarrollar habilidades en visualización de datos utilizando gráficos básicos, gráficos de dispersión y histogramas con Matplotlib.
- Aprender a manipular imágenes utilizando las funcionalidades de Matplotlib.

Metodología:

- Cada tema se introducirá con una breve lección que destaque los principios teóricos.
- Los participantes seguirán con ejercicios prácticos supervisados para reforzar los conceptos.
- Se fomentará el trabajo en equipo para el desarrollo de habilidades colaborativas y de comunicación.
- Los ejercicios serán seguidos por una discusión grupal para compartir enfoques y soluciones.
- Demostraciones en vivo y revisión de código colectiva ayudarán a ilustrar las mejores prácticas.

Recursos Necesarios:

- Computadoras con acceso a Python, NumPy y Matplotlib instalados.
- Documentación y recursos de aprendizaje en línea para consulta adicional.
- Proyector y pantalla para visualizar las presentaciones y demostraciones del instructor.

Evaluación:

- Revisión continua de los ejercicios prácticos para validar la comprensión y el progreso.
- Una evaluación final que consistirá en la creación de un script completo que integre los conceptos aprendidos.
- Retroalimentación y crítica constructiva serán proporcionadas para cada ejercicio y proyecto.

Contenido del Taller:

1. **Creación y Manipulación de Arrays:** Crea un array A con valores del 1 al 15. Luego, redimensiona este array a una matriz de 3x5.
2. **Operaciones Básicas:** Calcula la suma, la media y el producto de los elementos de A.
3. **Acceso y Slicing.** Selecciona el segundo y tercer elemento de la segunda fila de A.
4. **Indexación Booleana.** Crea un array B que contenga solo los elementos de A que son mayores que 7.
5. **Álgebra Lineal:** Calcula el determinante y la inversa de una matriz cuadrada C de 3x3 que crees.
6. **Estadísticas con NumPy.** Dado un array D de 100 números aleatorios, calcula su valor máximo, mínimo, media y desviación estándar.
7. **Gráfico Básico:** Grafica la función seno y coseno en el rango de -2π a 2π en el mismo gráfico.
8. **Gráficos de Dispersión:** Usa D para crear un gráfico de dispersión, representando los valores de D contra sus índices.
9. **Histogramas.** Crea un histograma de D, ajustando el número de bins para una mejor visualización.
10. **Manipulación de Imágenes con Matplotlib:** Lee una imagen con Matplotlib, conviértela a escala de grises y muestra ambas imágenes (original y en escala de grises).

Taller número 7. Aproximación de Funciones con Series de Taylor en Python

Descripción del Taller: Este taller introduce las Series de Taylor y su aplicación para la aproximación de funciones utilizando Python y NumPy. Los participantes desarrollarán funciones para calcular las aproximaciones de la exponencial, funciones trigonométricas y logaritmo natural, y visualizarán los resultados con Matplotlib. Este enfoque práctico no solo reforzará los conceptos matemáticos, sino que también mejorará las habilidades de programación de los participantes.

Objetivos del Taller:

- Comprender el concepto y la importancia de las Series de Taylor en matemáticas y análisis numérico.
- Implementar las Series de Taylor para funciones estándar usando Python y NumPy.
- Aprender a visualizar y comparar las aproximaciones de funciones con sus contrapartes estándar.
- Mejorar la habilidad de utilizar la programación para resolver problemas matemáticos complejos.

Metodología:

- Explicaciones teóricas para introducir cada concepto seguido de ejemplos ilustrativos.
- Sesiones de codificación en vivo donde los participantes implementan las series de Taylor paso a paso.
- Uso de Matplotlib para graficar las funciones y comparar las aproximaciones con las funciones reales.
- Discusiones grupales para analizar los resultados y comprender las diferencias.

Recursos Necesarios:

- Computadoras con Python, NumPy y Matplotlib instalados.
- Acceso a la documentación de Python y referencias matemáticas para las Series de Taylor.
- Materiales de referencia adicionales en línea para la teoría de las series y ejemplos de uso.

Evaluación:

- Evaluación de la precisión de las funciones implementadas mediante comparación con las funciones de NumPy.
- Revisión de las gráficas para asegurar la correcta visualización de los resultados.
- Participación activa en la discusión y análisis de los resultados de la graficación.

Contenido del Taller:

Una serie de Taylor es una representación de una función como una suma infinita de términos calculados a partir de los valores de sus derivadas en un solo punto. Es una herramienta poderosa en matemáticas para aproximar funciones complicadas con polinomios simples, especialmente cerca del punto alrededor del cual se construye la serie. Cada término de la serie es un producto de una derivada de la función evaluada en el punto y una potencia de la diferencia entre la variable y el punto, dividido por el factorial del número de la derivada.

$$e^x = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \ldots$$

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \ldots$$

$$\cos x = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots$$

$$\ln(1+x) = x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \ldots \quad for|x| < 1$$

$$\tan^{-1}(x) = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} + \ldots \quad for|x| < 1$$

Emplee Python junto con la biblioteca NumPy para desarrollar sus propias versiones de las funciones exponencial, seno, coseno, tangente y logaritmo natural mediante el uso de series de Taylor. Luego, aproveche Matplotlib para graficar todas estas funciones en un mismo subplot, facilitando así una comparativa visual directa entre ellas.