

PYTHON

RESUMEN

Tipos de datos fundamentales:

```
# Números
entero = 10
flotante = 3.14
complejo = 2 + 3j

# Cadenas de texto
cadena = "Hola, mundo!"

# Booleanos
verdadero = True
falso = False
```

Entrada/Salida básica

```
nombre = input("Ingrese su nombre: ")
print("Hola,", nombre)
```

Comentarios

```
# Esto es un comentario de una sola línea

"""
Esto es un comentario de varias líneas.
Se utiliza para hacer anotaciones extensas en el código.
"""
```

Operadores básicos

```
# Aritméticos
suma = 10 + 5
resta = 10 - 5
multiplicacion = 10 * 5
division = 10 / 5
modulo = 10 % 3
exponente = 10 ** 2

# Comparación
igualdad = 10 == 5
desigualdad = 10 != 5
mayor_que = 10 > 5
```

Estructuras de control

```
# Condicionales
if edad >= 18:
    print("Eres mayor de edad")
elif edad < 18:
    print("Eres menor de edad")
else:
    print("Edad no válida")

# Bucles
for i in range(5):
    print(i)

contador = 0
while contador < 5:
    print(contador)
    contador += 1
```

Funciones

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

resultado = suma(10, 5)
print(resultado)
```

Módulos y paquetes

```
# Importar un módulo
import math

# Utilizar funciones del módulo
raiz_cuadrada = math.sqrt(16)
print(raiz_cuadrada)
```

Manejo de excepciones

```
try:
    resultado = 10 / 0
except ZeroDivisionError:
    print("¡Error! División por cero.")
else:
    print("La división se realizó correctamente.")
```

Expresiones Regulares:

Las expresiones regulares te permiten buscar y manipular patrones de texto.

```
import re

texto = "Hola, 123 mundo! 456"
patron = r'\d+'
resultados = re.findall(patron, texto)
print(resultados) # Salida: ['123', '456']
```

Manipulación de Fechas y Tiempo:

El módulo datetime te permite trabajar con fechas y tiempos de manera fácil y eficiente.

```
from datetime import datetime, timedelta

ahora = datetime.now()
print(ahora.strftime("%Y-%m-%d %H:%M:%S"))

nueva_fecha = ahora + timedelta(days=7)
print(nueva_fecha.strftime("%Y-%m-%d"))
```

Cadenas

```
# Iterar sobre una cadena letra por letra
cadena = "Python"

print("Recorriendo la cadena letra por letra:")
for letra in cadena:
    print(letra)

# Concatenación de Cadenas
cadena1 = "Hola"
cadena2 = "mundo"
concatenada = cadena1 + " " + cadena2
print(concatenada) # Salida: "Hola mundo"

palabras = ["Hola", "mundo"]
concatenada = " ".join(palabras)
print(concatenada) # Salida: "Hola mundo"

# Longitud de una Cadena
cadena = "Python"
longitud = len(cadena)
print(longitud) # Salida: 6

# Acceso a Caracteres por Índice
cadena = "Python"
primer_caracter = cadena[0] # Índice 0
ultimo_caracter = cadena[-1] # Último índice
print(primer_caracter, ultimo_caracter) # Salida: "P" "n"
```

```

# Slicing de Cadenas
cadena = "Python"
subcadena = cadena[1:4] # Desde el índice 1 hasta el 3 (no incluido)
print(subcadena) # Salida: "yth"

subcadena = cadena[:3] # Desde el principio hasta el índice 2
print(subcadena) # Salida: "Py"

subcadena = cadena[3:] # Desde el índice 3 hasta el final
print(subcadena) # Salida: "hon"

# Verificación de Subcadenas
cadena = "Python es genial"
subcadena = "genial"
if subcadena in cadena:
    print("La subcadena está presente.")
else:
    print("La subcadena no está presente.")

# Formateo de Cadenas
nombre = "Juan"
edad = 30
print("Hola, me llamo {} y tengo {} años.".format(nombre, edad))
# Salida: "Hola, me llamo Juan y tengo 30 años."

print(f"Hola, me llamo {nombre} y tengo {edad} años.")
# Salida: "Hola, me llamo Juan y tengo 30 años."

# Búsqueda y Reemplazo
cadena = "Python es un lenguaje de programación"
indice = cadena.find("lenguaje")
print("Índice de 'lenguaje':", indice) # Salida: 11

nueva_cadena = cadena.replace("Python", "JavaScript")
print("Nueva cadena:", nueva_cadena)
# Salida: "JavaScript es un lenguaje de programación"

# Verificación de Tipo de Caracteres
cadena1 = "Python"
print(cadena1.isalpha()) # Salida: True

cadena2 = "123"
print(cadena2.isdigit()) # Salida: True

cadena3 = "Python3"
print(cadena3.isalnum()) # Salida: True

cadena4 = "   "
print(cadena4.isspace()) # Salida: True

# Partición de Cadenas
cadena = "Python es un lenguaje de programación"
partes = cadena.partition("un")
print(partes) # Salida: ('Python es ', 'un', ' lenguaje de programación')

```

```

# Función split()
texto = "Hola mundo"
palabras = texto.split() # Por defecto, divide por espacio en blanco
print(palabras) # Salida: ['Hola', 'mundo']

texto = "manzana,banana,cereza"
frutas = texto.split(",")
print(frutas) # Salida: ['manzana', 'banana', 'cereza']

# Función join()
palabras = ['Hola', 'mundo']
texto = ' '.join(palabras)
print(texto) # Salida: "Hola mundo"

frutas = ['manzana', 'banana', 'cereza']
texto = ','.join(frutas)
print(texto) # Salida: "manzana,banana,cereza"

# Funciones strip(), lstrip() y rstrip()
texto = "  Hola mundo  "
limpio = texto.strip()
print(limpio) # Salida: "Hola mundo"

texto = "---Hola mundo---"
limpio = texto.strip('-')
print(limpio) # Salida: "Hola mundo"

texto = "  Hola mundo  "
izquierda = texto.lstrip()
print(izquierda) # Salida: "Hola mundo  "

texto = "  Hola mundo  "
derecha = texto.rstrip()
print(derecha) # Salida: "  Hola mundo"

# Funciones lower() y upper()
texto = "Hola Mundo"
minusculas = texto.lower()
print(minusculas) # Salida: "hola mundo"

texto = "Hola Mundo"
mayusculas = texto.upper()
print(mayusculas) # Salida: "HOLA MUNDO"

# Función replace()
texto = "Hola mundo"
nuevo_texto = texto.replace("mundo", "Python")
print(nuevo_texto) # Salida: "Hola Python"

# Funciones startswith() y endswith()
texto = "Hola mundo"
print(texto.startswith("Hola")) # Salida: True
print(texto.endswith("mundo")) # Salida: True

```

Estructuras de datos:

```
# Listas
lista = [1, 2, 3, "cuatro", 5.5]

# Tuplas
tupla = (1, 2, 3, "cuatro", 5.5)

# Conjuntos
conjunto = {1, 2, 3, 4, 5}

# Diccionesarios
diccionario = {"clave1": "valor1", "clave2": "valor2", "clave3": "valor3"}
```

Listas:

```
# Crear una lista
lista = [1, 2, 3, 4, 5]

# Agregar un elemento al final de la lista
lista.append(6)

# Extender la lista agregando los elementos de otro iterable
lista.extend([7, 8, 9])

# Insertar un elemento en una posición específica
lista.insert(2, 10)

# Eliminar la primera ocurrencia de un elemento específico
lista.remove(3)

# Eliminar y devolver el elemento en la posición dada
elemento_eliminado = lista.pop(0)

# Encontrar el índice de la primera aparición de un elemento
indice = lista.index(4)

# Contar cuántas veces aparece un elemento en la lista
conteo = lista.count(2)

# Ordenar la lista en su lugar
lista.sort()

# Invertir el orden de los elementos en la lista en su lugar
lista.reverse()
```

Tuplas : Las tuplas son inmutables

```
# Crear una tupla
tupla = (1, 2, 3, 4, 5)

# Indexación
primer_elemento = tupla[0]      # Acceder al primer elemento
ultimo_elemento = tupla[-1]     # Acceder al último elemento

# Slicing
# Obtener sub-tupla desde el segundo hasta el cuarto elemento
sub_tupla1 = tupla[1:4]
# Obtener sub-tupla desde el tercer elemento hasta el final
sub_tupla2 = tupla[2:]
# Obtener sub-tupla desde el principio hasta el tercer elemento
sub_tupla3 = tupla[:3]

# Longitud de una tupla
longitud = len(tupla)           # Longitud de la tupla

# Iteración
# Iterar sobre los elementos de la tupla
for elemento in tupla:
    print(elemento)

# Concatenación de tuplas
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla_concatenada = tupla1 + tupla2  # Concatenar dos tuplas

# Repetición de tuplas
tupla = (1, 2)
tupla_repetida = tupla * 3           # Repetir una tupla

# Comprobar si un elemento está en la tupla
if 3 in tupla:
    print("El elemento 3 está en la tupla")
```

Conjuntos

```
# Crear un conjunto
conjunto = {1, 2, 3, 4, 5}

# Agregar un elemento al conjunto
conjunto.add(6)

# Eliminar un elemento específico del conjunto
conjunto.remove(3)

# Eliminar un elemento del conjunto si está presente
conjunto.discard(7)

# Eliminar y devolver un elemento aleatorio del conjunto
elemento_eliminado = conjunto.pop()

# Limpiar el conjunto, eliminando todos los elementos
conjunto.clear()

# Unir dos conjuntos
conjunto.union({6, 7, 8})

# Intersección de dos conjuntos
conjunto.intersection({4, 5, 6})

# Diferencia de dos conjuntos
conjunto.difference({1, 2, 3})

# Diferencia simétrica de dos conjuntos
conjunto.symmetric_difference({3, 4, 5})
```


Diccionarios

```
# Crear un diccionario
diccionario = {'a': 1, 'b': 2, 'c': 3}

# Acceder a un valor por clave
valor = diccionario['a']
print("Valor de 'a':", valor) # Salida: 1

# Modificar un valor
diccionario['b'] = 20

# Agregar un nuevo elemento
diccionario['d'] = 4

# Eliminar un elemento por clave utilizando pop()
valor_eliminado = diccionario.pop('c')
print("Valor eliminado de 'c':", valor_eliminado) # Salida: 3
print("Diccionario actualizado:", diccionario) # Salida: {'a': 1, 'b': 20, 'd': 4}

# Verificar si una clave está en el diccionario
if 'b' in diccionario:
    print("'b' está en el diccionario.")
else:
    print("'b' no está en el diccionario.")

# Iterar sobre claves y valores
for clave, valor in diccionario.items():
    print(clave, valor)

# Método get para obtener un valor por clave (si la clave no existe, devuelve None o
valor = diccionario.get('a')
print("Valor de 'a':", valor) # Salida: 1

# Método keys para obtener todas las claves del diccionario
claves = diccionario.keys()
print("Claves del diccionario:", claves) # Salida: dict_keys(['a', 'b', 'd'])

# Método values para obtener todos los valores del diccionario
valores = diccionario.values()
print("Valores del diccionario:", valores) # Salida: dict_values([1, 20, 4])

# Método items para obtener pares clave-valor como tuplas
items = diccionario.items()
print("Pares clave-valor del diccionario:", items) # Salida: dict_items([('a', 1), ('b', 20), ('d', 4)])
```

Programación Funcional:

La programación funcional en Python implica el uso de funciones de orden superior, lambdas, map, filter y reduce.

```
# Map
numeros = [1, 2, 3, 4, 5]
cuadrados = list(map(lambda x: x ** 2, numeros))
print(cuadrados) # Salida: [1, 4, 9, 16, 25]

# Filter
pares = list(filter(lambda x: x % 2 == 0, numeros))
print(pares) # Salida: [2, 4]

# Reduce
from functools import reduce
suma = reduce(lambda x, y: x + y, numeros)
print(suma) # Salida: 15
```

Generadores y Iteradores:

Los generadores y los iteradores son útiles para trabajar con secuencias de datos.

```
# Generador
def generador():
    yield 1
    yield 2
    yield 3

for valor in generador():
    print(valor) # Salida: 1, 2, 3

# Iterador
iterador = iter([1, 2, 3])
print(next(iterador)) # Salida: 1
print(next(iterador)) # Salida: 2
```

Decoradores y Context Managers:

Los decoradores y los context managers son herramientas poderosas para manipular el comportamiento de las funciones y el contexto de ejecución.

```
# Decorador
def decorador(funcion):
    def wrapper():
        print("Antes de la llamada a la función")
        funcion()
        print("Después de la llamada a la función")
    return wrapper

@decorador
def saludar():
    print("Hola!")

saludar()

# Context Manager
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()
```

Menú

```
def mostrar_menu():
    print("Bienvenido al menú:")
    print("1. Opción 1")
    print("2. Opción 2")
    print("3. Opción 3")
    print("4. Salir")

def procesar_opcion(opcion):
    if opcion == "1":
        print("Seleccionaste la Opción 1")
    elif opcion == "2":
        print("Seleccionaste la Opción 2")
    elif opcion == "3":
        print("Seleccionaste la Opción 3")
    elif opcion == "4":
        print("Saliendo del programa...")
    else:
        print("Opción no válida")

# Bucle principal del menú
while True:
    mostrar_menu()
    opcion = input("Ingrese el número de la opción que desea: ")

    if opcion == "4":
        break # Salir del bucle si se selecciona la opción 4
    else:
        procesar_opcion(opcion)
```

ARCHIVOS Y DIRECTORIOS

Manejo de Archivos y Directorios:

Python proporciona funciones y módulos para trabajar con archivos y directorios de forma eficiente.

```
import os

# Listar archivos en un directorio
archivos = os.listdir('.')
print(archivos)

# Crear un directorio
os.mkdir('nuevo_directorio')

# Borrar un archivo
os.remove('archivo_a_borrar.txt')
```

Ficheros: Leer y escribir en ficheros

```
# Para leer archivos:

# Abrir un archivo en modo lectura
with open("archivo.txt", "r") as archivo:
    contenido = archivo.read()

# Leer líneas del archivo
with open("archivo.txt", "r") as archivo:
    lineas = archivo.readlines() # Devuelve una lista con todas las líneas del archivo
    for linea in lineas:
        print(linea)

# Para escribir en archivos:

# Abrir un archivo en modo escritura
with open("archivo.txt", "w") as archivo:
    archivo.write("Hola, mundo!\n")
    archivo.write("Segunda línea\n")

# Agregar contenido a un archivo sin eliminar el contenido existente
with open("archivo.txt", "a") as archivo:
    archivo.write("Otra línea\n")
```

JSON

```
import json

# Para leer un archivo JSON:

# Abrir el archivo JSON en modo lectura
with open("datos.json", "r") as archivo:
    datos = json.load(archivo)

# Ahora 'datos' contiene la información del archivo JSON como un diccionario de Python
print(datos)

# Para escribir en un archivo JSON:

# Datos que queremos escribir en el archivo JSON
datos = {
    "nombre": "Juan",
    "edad": 30,
    "ciudad": "Madrid"
}

# Abrir el archivo JSON en modo escritura
with open("datos.json", "w") as archivo:
    json.dump(datos, archivo)
```

CONCURRENCIA

Tenemos una tarea que queremos realizar de forma paralela en varios hilos, y luego queremos sincronizar esos hilos para que esperen hasta que todos hayan terminado antes de continuar. Por ejemplo, vamos a simular la descarga de varios archivos simultáneamente y luego fusionar los resultados.

```
import threading
import time

# Función para simular la descarga de un archivo
def descargar_archivo(nombre_archivo):
    print(f"Iniciando descarga de {nombre_archivo}")
    time.sleep(2) # Simulamos la descarga esperando 2 segundos
    print(f"{nombre_archivo} descargado")

# Lista de archivos a descargar
archivos = ["archivo1.txt", "archivo2.txt", "archivo3.txt"]

# Lista para almacenar los hilos
hilos = []

# Creamos un hilo para cada archivo
for archivo in archivos:
    hilo = threading.Thread(target=descargar_archivo, args=(archivo,))
    hilos.append(hilo)
    hilo.start()

# Esperamos a que todos los hilos terminen
for hilo in hilos:
    hilo.join()

print("Descargas completadas. Fusionando resultados...")
# Aquí se podría realizar alguna operación con los archivos descargados
```

- Definimos una función `descargar_archivo()` que simula la descarga de un archivo. Esta función espera 2 segundos para simular el tiempo de descarga.
- Creamos una lista de nombres de archivos a descargar.
- Creamos un hilo para cada archivo en la lista utilizando la función `Thread()` del módulo `threading`. Especificamos la función a ejecutar en el hilo (en este caso, `descargar_archivo`) y pasamos los argumentos necesarios a través del parámetro `args`.
- Iniciamos cada hilo utilizando el método `start()`.
- Utilizamos el método `join()` en cada hilo para esperar a que todos los hilos hayan terminado antes de continuar con la ejecución del programa principal.

- Una vez que todos los hilos han terminado, el programa continúa con la operación de fusión de resultados (en este caso, simplemente se imprime un mensaje indicando que las descargas han sido completadas).

Con **asyncio** puedes lograr el mismo efecto pero de una manera asincrónica. Aquí tienes un ejemplo detallado y comentado de cómo usar asyncio para realizar descargas de archivos de forma simultánea y luego fusionar los resultados:

```
import asyncio

# Función asincrónica para simular la descarga de un archivo
async def descargar_archivo(nombre_archivo):
    print(f"Iniciando descarga de {nombre_archivo}")
    await asyncio.sleep(2) # Simulamos la descarga esperando 2 segundos
    print(f"{nombre_archivo} descargado")

# Lista de archivos a descargar
archivos = ["archivo1.txt", "archivo2.txt", "archivo3.txt"]

# Función principal asincrónica para manejar las descargas
async def descargar_archivos():
    # Creamos una tarea asincrónica para cada archivo
    tareas = [asyncio.create_task(descargar_archivo(archivo)) for archivo in archivos]
    # Esperamos a que todas las tareas se completen
    await asyncio.gather(*tareas)
    print("Descargas completadas. Fusionando resultados...")
    # Aquí se podría realizar alguna operación con los archivos descargados

# Ejecutamos la función principal asincrónica
asyncio.run(descargar_archivos())
```

BASES DE DATOS

Conexión a base de datos Mysql

```
pip install mysql-connector-python

import mysql.connector

# Establecer la conexión con la base de datos
conexion = mysql.connector.connect(
    host="localhost",
    user="tu_usuario",
    password="tu_contraseña",
    database="nombre_de_tu_base_de_datos"
)

# Crear un cursor para ejecutar consultas
cursor = conexion.cursor()


# Consulta parametrizada
consulta = "SELECT * FROM tabla WHERE columna = %s"
valor = ("valor_a_buscar",) # Debe ser una tupla, incluso si solo hay un valor
cursor.execute(consulta, valor)

# Obtener los resultados de la consulta
resultados = cursor.fetchall()
for fila in resultados:
    print(fila)

# Insertar datos
consulta_insert = "INSERT INTO tabla (columna1, columna2) VALUES (%s, %s)"
datos_insert = ("valor_columna1", "valor_columna2")
cursor.execute(consulta_insert, datos_insert)

# Confirmar los cambios en la base de datos
conexion.commit()

# Cerrar el cursor y la conexión
cursor.close()
conexion.close()
```



La consulta parametrizada se realiza usando %s como marcador de posición. Luego, pasas los valores a la función execute() como una tupla.

Para insertar datos, también usas %s como marcadores de posición y luego pasas los valores a la función execute() como una tupla.

Después de insertar datos, es importante confirmar los cambios en la base de datos utilizando commit(). Finalmente, cierra el cursor y la conexión

Integración con Bases de Datos NoSQL:

```
pip install pymongo

import pymongo

# Conexión a MongoDB
cliente = pymongo.MongoClient("mongodb://localhost:27017/")

# Seleccionar una base de datos (si no existe, se creará automáticamente)
db = cliente["mi_base_de_datos"]

# Seleccionar una colección (si no existe, se creará automáticamente)
coleccion = db["mi_coleccion"]

# Insertar un solo documento
documento = {"nombre": "Juan", "edad": 30}
resultado = coleccion.insert_one(documento)
print("ID del documento insertado:", resultado.inserted_id)

# Insertar múltiples documentos
documentos = [
    {"nombre": "Ana", "edad": 25},
    {"nombre": "Carlos", "edad": 35}
]
resultado = coleccion.insert_many(documentos)
print("IDs de los documentos insertados:", resultado.inserted_ids)

# Consulta todos los documentos en la colección
print("Todos los documentos en la colección:")
resultados = coleccion.find()
for documento in resultados:
    print(documento)

# Consulta con filtro
filtro = {"nombre": "Juan"}
print("Documentos con nombre 'Juan':")
resultados = coleccion.find(filtro)
for documento in resultados:
    print(documento)

# Actualizar datos
filtro = {"nombre": "Juan"}
nuevos_valores = {"$set": {"edad": 31}}
coleccion.update_one(filtro, nuevos_valores)
print("Datos actualizados.")

# Eliminar datos
filtro = {"nombre": "Juan"}
coleccion.delete_one(filtro)
print("Documento con nombre 'Juan' eliminado.")
```

Pruebas Unitarias y Testing:

Las pruebas unitarias son fundamentales para garantizar el correcto funcionamiento del código.

```
import unittest

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

class TestSuma(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(suma(3, 4), 7)

if __name__ == '__main__':
    unittest.main()
```

ORIENTACIÓN A OBJETOS

Clases y Objetos:

Una clase es un plano para crear objetos, define propiedades y comportamientos.

Un objeto es una instancia de una clase.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

# Crear un objeto (instancia de la clase Persona)
persona1 = Persona("Juan", 30)
```

Atributos y Métodos:

Los atributos son variables que pertenecen a la clase u objeto.

Los métodos son funciones que pertenecen a la clase u objeto.

```
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def saludar(self):
        print("Hola, mi nombre es", self.nombre)

persona1 = Persona("Juan", 30)
persona1.saludar() # Salida: Hola, mi nombre es Juan
```

Encapsulamiento:

Encapsulamiento es el proceso de ocultar los detalles internos de un objeto.

Se utiliza para proteger los datos de una clase y ocultar la implementación interna.

```
class Persona:
    def __init__(self, nombre, edad):
        self._nombre = nombre # Atributo protegido
        self.__edad = edad    # Atributo privado

persona1 = Persona("Juan", 30)
print(persona1._nombre) # Acceso a un atributo protegido
# print(persona1.__edad) # Esto generará un error
```

Herencia:

La herencia permite que una clase herede atributos y métodos de otra clase.

```
class Empleado(Persona):
    def __init__(self, nombre, edad, salario):
        super().__init__(nombre, edad)
        self.salario = salario

empleado1 = Empleado("Ana", 25, 50000)
print(empleado1.nombre) # Acceso al atributo heredado
```

Polimorfismo:

El polimorfismo permite que objetos de diferentes clases respondan al mismo método.

```
class Animal:
    def sonido(self):
        pass

class Perro(Animal):
    def sonido(self):
        print("Guau")

class Gato(Animal):
    def sonido(self):
        print("Miau")

def hacer_sonar(animal):
    animal.sonido()

perro = Perro()
gato = Gato()
hacer_sonar(perro) # Salida: Guau
hacer_sonar(gato) # Salida: Miau
```

Métodos Especiales (Magic Methods):

Permiten definir el comportamiento especial de los objetos en Python.

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

punto = Punto(3, 4)
print(punto) # Salida: (3, 4)
```

Decoradores de Clases:

Permiten modificar el comportamiento de las clases y sus métodos.

```
def decorador(clase):
    class NuevaClase(clase):
        def nuevo_metodo(self):
            print("Nuevo método añadido")
    return NuevaClase

@decorador
class MiClase:
    def metodo_existente(self):
        print("Método existente")

objeto = MiClase()
objeto.metodo_existente() # Salida: Método existente
objeto.nuevo_metodo()     # Salida: Nuevo método añadido
```

En Python, no se pueden tener varios métodos con el mismo nombre, incluidos los constructores (`__init__`). Sin embargo, puedes lograr un comportamiento similar utilizando argumentos con valores predeterminados en el método `__init__`, lo que te permite crear instancias de la clase con diferentes conjuntos de argumentos.

Cómo usar argumentos con valores predeterminados para simular la presencia de múltiples constructores:

```
class Tarea:
    def __init__(self, nombre, completada=False):
        self.nombre = nombre
        self.completada = completada

# Crear una tarea con el estado de completada predeterminado (False)
tarea1 = Tarea("Limpiar")
print(tarea1.nombre, tarea1.completada) # Imprimirá: Limpiar False

# Crear una tarea con el estado de completada definido como True
tarea2 = Tarea("Recoger", True)
print(tarea2.nombre, tarea2.completada) # Imprimirá: Recoger True
```

En Python, los atributos de una clase pueden ser declarados como inmutables utilizando la propiedad `@property` junto con un método getter, pero sin un método setter. Esto significa que una vez que el valor del atributo se establece durante la inicialización de la instancia, no se puede modificar directamente desde fuera de la clase.

```
class Persona:
    def __init__(self, nombre, edad):
        self._nombre = nombre
        self._edad = edad

    @property
    def nombre(self):
        return self._nombre

    @property
    def edad(self):
        return self._edad

# Crear una instancia de la clase Persona
persona = Persona("Juan", 30)

# Intentar cambiar el nombre y la edad (lo cual debería fallar)
# Esto generará un AttributeError porque no hay un método setter para nombre y edad
try:
    persona.nombre = "Pedro"
    persona.edad = 25
except AttributeError as e:
    print("Error:", e)
```

PANDAS

Pandas es una biblioteca de Python muy popular y potente utilizada principalmente para análisis y manipulación de datos. Proporciona estructuras de datos flexibles y rápidas diseñadas para trabajar con datos tabulares y etiquetados de una manera intuitiva. Aquí tienes una descripción de los principales componentes y características de Pandas:

Componentes Principales:

DataFrame:

- Es la estructura de datos principal en Pandas.
- Se trata de una estructura de datos bidimensional, similar a una tabla o hoja de cálculo, donde los datos se organizan en filas y columnas.
- Cada columna en un DataFrame representa una variable, y cada fila representa una observación o muestra.

Series:

- Es una estructura de datos unidimensional que puede contener cualquier tipo de datos.
- Se puede considerar como una columna de un DataFrame.
- Las Series también pueden ser indexadas, lo que permite un acceso eficiente a los datos.

Características Clave:

Carga y Escritura de Datos:

- Pandas ofrece funciones para cargar y escribir datos desde y hacia una variedad de formatos de archivo, como CSV, Excel, SQL, JSON, HTML, etc.

Manipulación de Datos:

- Pandas proporciona numerosas funciones y métodos para manipular datos, como filtrado, selección, ordenación, agrupación, unión, concatenación, etc.

Limpieza de Datos:

- Permite limpiar y preprocesar datos, incluyendo manejo de valores nulos, eliminación de duplicados, conversión de tipos de datos, y más.

Análisis y Exploración de Datos:

- Ofrece herramientas para realizar análisis estadísticos básicos, resúmenes descriptivos, visualización de datos y más.

Operaciones Matemáticas y Estadísticas:

- Pandas facilita la aplicación de operaciones matemáticas y estadísticas a los datos, tanto a nivel de columna como de DataFrame.

Indexación y Selección Eficientes:

- Proporciona una potente funcionalidad de indexación y selección que permite acceder a los datos de manera eficiente.

Integración con otras Bibliotecas:

- Se integra bien con otras bibliotecas de Python, como NumPy, Matplotlib, Scikit-learn y más, lo que permite un análisis de datos más completo. PANDAS

Ejemplo de Pandas con .CSV

```
import pandas as pd

# Cargar datos desde el archivo CSV
df = pd.read_csv("ventas.csv")

# Mostrar las primeras filas del DataFrame
print("Primeras filas del DataFrame:")
print(df.head())

# Verificar tipos de datos y valores nulos
print("\nInformación del DataFrame:")
print(df.info())

# Convertir la columna "Fecha" a tipo datetime
df["Fecha"] = pd.to_datetime(df["Fecha"])

# Calcular el total de ventas por producto
ventas_por_producto = df.groupby("Producto")["Total"].sum().reset_index()
print("\nTotal de ventas por producto:")
print(ventas_por_producto)

# Calcular el total de ventas por mes
df["Mes"] = df["Fecha"].dt.month
ventas_por_mes = df.groupby("Mes")["Total"].sum().reset_index()
print("\nTotal de ventas por mes:")
print(ventas_por_mes)

# Exportar los resultados a un nuevo archivo CSV
ventas_por_producto.to_csv("ventas_por_producto.csv", index=False)
ventas_por_mes.to_csv("ventas_por_mes.csv", index=False)
```