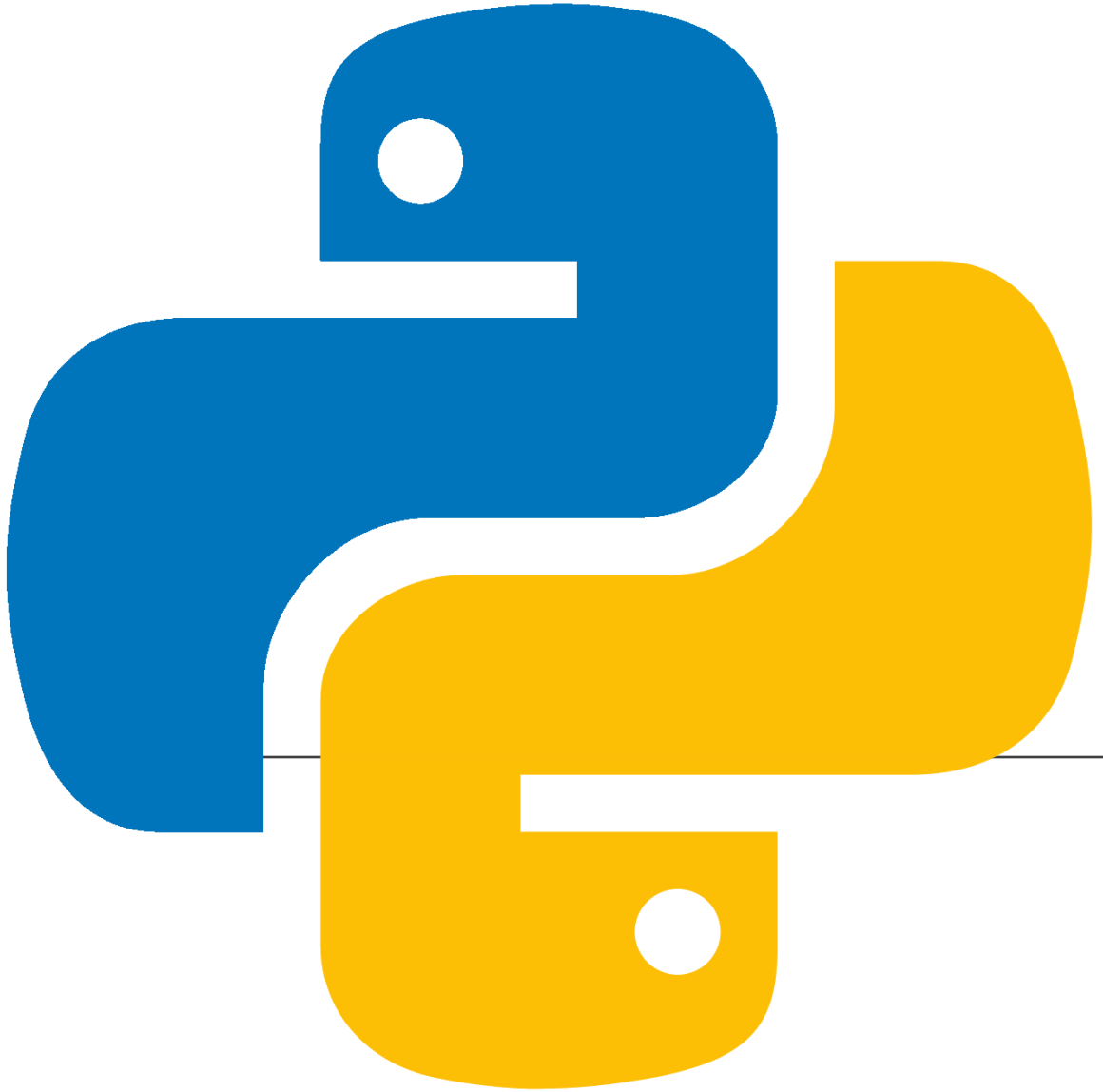


Python: Análisis de Datos y Ecosistemas Modernos de Analítica

TESTING E INTRODUCCIÓN AL ANÁLISIS DE DATOS

Recordando

El manejo eficiente de dependencias garantiza **consistencia** en el desarrollo, mientras que el uso de contextos aislados (como venv o Docker) asegura **reproducibilidad** en múltiples niveles.

Recordando

Herramientas de productividad como **Ruff** mejoran el flujo de desarrollo y la consistencia del código, mientras que **mypy** aporta garantías de verificación de tipos en el contexto dinámico de Python.

La integración de estas herramientas en plataformas de **CI/CD** permite automatizar los chequeos de calidad y aumentar la productividad de los equipos de desarrollo.

Encuesta

¿Por qué es útil el chequeo estático de tipos (mypy, etc.) en un lenguaje dinámico como Python?

Encuesta

¿Por qué es útil el chequeo estático de tipos (mypy, etc.) en un lenguaje dinámico como Python?

Aporta garantías adicionales, permite detectar errores de tipos de forma proactiva

TESTING

Verificando

Una parte fundamental de todo flujo de trabajo es la **verificación del código (testing)** y su integración en los **pipelines de CI/CD** para asegurar calidad y estabilidad en los proyectos.

Testing o Verificación

El **testing** consiste en evaluar el código bajo condiciones que aseguren que su comportamiento cumple con los **requisitos funcionales** establecidos.

En proyectos grandes los errores o *bug*s son inevitables, pero reducir su impacto es esencial: previene que afecten de forma crítica a los usuarios y ayuda a **disminuir los costos de mantenimiento** a largo plazo.

¿Reducir costos?

Contratar ingenieros o programadores implica un costo significativo, y dedicar ese tiempo únicamente a **corregir bugs** resulta una forma poco eficiente de invertir recursos.

Importancia

En industrias tradicionales, el **control de calidad** es evidente, como ocurre en la producción de alimentos. Pero en el software, los **riesgos potenciales** pueden ser igual o incluso más críticos.

- **Nissan** tuvo que retirar más de un millón de autos debido a fallas en el software de detección de accidentes.
- **Bloomberg Terminal**, utilizada por inversores, presentó un *glitch* que causó pérdidas económicas a numerosos clientes del mercado bursátil.
- Etc.

What is Software Testing? Definition, Basics & Types in Software Engineering (guru99.com)

Importancia

Mediante testing podemos detectar errores en la arquitectura del programa, malas decisiones, funcionalidad incorrecta, vulnerabilidades de seguridad, etc.

Buenas prácticas

Existen muchas **buenas prácticas** en el proceso de testing que ayudan a optimizar la verificación del software.

La más importante es la **automatización** de la mayor cantidad de pruebas posibles, especialmente en proyectos de gran escala.

Tipos de test

Tests funcionales

Verifican que el software cumpla con los **requisitos definidos**.
Por ejemplo, comprobar que una función retorne los valores esperados en distintos escenarios.

Los **tests de unidad** y los **tests de integración** son ejemplos de pruebas funcionales.

Tipos de test

Tests no funcionales

Evalúan si el **entorno** y las **capacidades del sistema** pueden soportar los requisitos funcionales.

Por ejemplo: comprobar que un servicio web resista una alta carga de usuarios simultáneos o que la aplicación ofrezca una experiencia de uso adecuada.

Encuesta

Comprobar que un usuario puede iniciar sesión en una aplicación con sus credenciales, y que el sistema rechaza credenciales inválidas. ¿Qué tipo de test es este?

Encuesta

Comprobar que un usuario puede iniciar sesión en una aplicación con sus credenciales, y que el sistema rechaza credenciales inválidas. ¿Qué tipo de test es este?

Funcional, estos verifican que el software cumpla con los requisitos definidos.

Tipos de test

En la práctica, no existe una única forma de agrupar los tipos de tests.

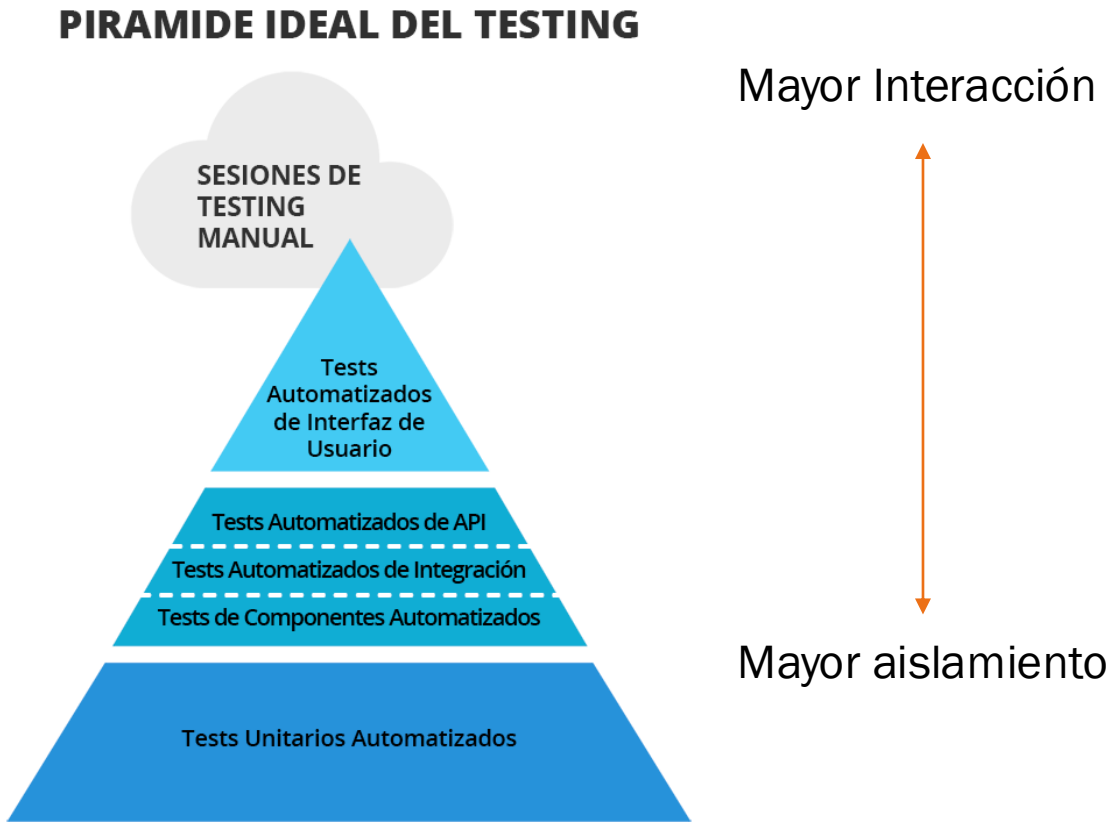
Existen múltiples formas de organizarlos, dependiendo del **contexto** y de los **objetivos** del proyecto.

Different Types of Testing in Software: 100 Examples (guru99.com)

Pirámide: Costo vs Cantidad

La **pirámide del testing** es un modelo que organiza los distintos tipos de pruebas según su alcance, costo y velocidad de ejecución.

Mientras más alto en la pirámide mayor cobertura, pero también mayor costo.



¿Por qué diferenciar tipos?

No todos los tests ofrecen el mismo **costo/beneficio**.

Los **tests unitarios** son baratos y rápidos, detectan errores en funciones pequeñas pero no validan el sistema completo; los **tests de integración** tienen un costo medio, verifican la interacción entre módulos y requieren más tiempo; y los **tests end-to-end (E2E)** son los más caros, simulan escenarios completos de usuario, ofrecen mayor cobertura pero son lentos y difíciles de mantener.

Objetivo

Tener muchos tests unitarios rápidos, una cantidad moderada de integración y pocos end-to-end, logrando así un balance eficiente de cobertura y costo.

La **estrategia de testing** siempre es relativa al **proyecto** y a los **recursos disponibles**.
Cada equipo debe equilibrar cobertura, costo y velocidad según sus necesidades y prioridades.

Tests funcionales

Es importante aplicar distintos tipos de pruebas en una aplicación, pero nos centraremos en dos:

- Los **tests de unidad**, que evalúan el comportamiento aislado de una función o componente específico

- **Los tests de integración**, que validan cómo interactúan correctamente varias unidades cuando trabajan en conjunto.

Testing en Python

Existen muchas librerías para evaluar aplicaciones en Python, pero la más común para pruebas de unidad e integración es **pytest**.

Cuenta con un sistema de **plugins** que permite extender sus capacidades según la necesidad, por ejemplo para integrarse fácilmente con aplicaciones desarrolladas en **Flask,** o modelos usando **pandas**.

Testing en Python: assert

assert es una instrucción de Python que verifica que una condición sea verdadera; si no lo es, genera un **error** durante la ejecución.

```python
def test_siempre_pasa():
    assert True

def test_siempre_falla():
    assert False
```

Effective Python Testing With Pytest – Real Python

Pytest: requisitos

pytest identifica automáticamente qué funciones son pruebas según su nombre y el del módulo:

- El archivo debe comenzar con test_.
- Cada función de prueba también debe comenzar con test_.

Este archivo será detectado

```
# contenido test_ejemplo.py
def incrementar(x):
    return x + 1


def test_incremento():
    assert incrementar(3) == 5
```

Este archivo no será detectado

```
# contenido ejemplo.py
def incrementar(x):
    return x + 1


def test_incremento():
    assert incrementar(3) == 5
```

pytest: helps you write better programs — pytest documentation

Pytest: estructura

La librería **pytest** no impone una estructura fija, siempre que pueda descubrir los archivos de prueba.

Sin embargo, la organización puede variar según el proyecto.

Una práctica muy común es crear una carpeta llamada **tests/**, donde se ubican los archivos de prueba, cada uno nombrado con el prefijo test_.

pytest: ejecutar los tests

Una vez instalada la librería, las pruebas pueden ejecutarse con **pytest** desde la carpeta principal del proyecto. Esto puede hacerse mediante comandos en la terminal, instrucciones de automatización en un pipeline de CI/CD, o directamente desde el **IDE** si ofrece soporte integrado.

```
python -m pytest
```

```
============================================================ test session starts
============================================================
platform win32 -- Python 3.10.5, pytest-7.1.2, pluggy-1.0.0
rootdir: C:\Users\deort\OneDrive\Documents\GitHub\ista-python-curso-2022
collected 1 item

escuela_api\tests\test_estudiantes.py .                                  [100%]

============================================================ 1 passed in 0.04s
============================================================
```

Encuesta

¿Cuál es la función principal de assert en las pruebas unitarias??

Encuesta

¿Cuál es la función principal de assert en las pruebas unitarias??

Verificar que una condición es verdadera

Valor de verdad

En Python, cualquier valor puede evaluarse en un **contexto booleano** (por ejemplo, en un if, un while, o en un assert).

El intérprete no solo evalúa True o False directamente, sino que aplica reglas de **truthiness**: decide si un objeto debe considerarse **verdadero** (*truthy*) o **falso** (*falsy*).

Valor de verdad

En Python, un assert, un condicional (if, elif) o un loop (while, for) evalúan cualquier valor que sea **truthy**, aunque no sea estrictamente un True o False.

```
if "hola":      # string no vacío → truthy
    print("Verdadero")


if "":          # string vacío → falsy
    print("Nunca se ejecuta")
```

No todo lo vacío es estrictamente `False`, pero muchas estructuras vacías (listas, strings, diccionarios, sets) son tratadas como **falsy**.

Valor de verdad

Cuando usamos assert o condiciones, confiar solo en truthiness puede ser **ambiguo.**

```
valor = 0
assert not valor     # Confuso, porque 0 es falsy
```

¿Como diferencio entre 0 y None? Ambos son *falsy*, pero representan cosas muy distintas.

En estos casos es mejor ser **explícito** al validar, para evitar confusión y reflejar la intención real del código.

```
assert valor == 0         # Chequea que sea cero
assert resultado is None  # Chequea que sea None
```

Valor de verdad por defecto

Valor	Evaluación
`0, 0.0, 0j`	Falsy
`""` (string vacío)	Falsy
`[]` (lista vacía)	Falsy
`{}` (diccionario vacío)	Falsy
`set()` (conjunto vacío)	Falsy
`None`	Falsy
`False`	Falsy
Cualquier otro valor distinto	Truthy

Encuesta

Queremos validar que una función obtener_precio() devuelva un valor correcto.

¿Cuál es el problema con este código?

```
precio = obtener_precio()
assert precio
```

Encuesta

Queremos validar que una función obtener_precio() devuelva un valor correcto.

¿Cuál es el problema con este código?

```
precio = obtener_precio()
assert precio
```

El valor 0 produce un error, a pesar de ser un precio correcto.

Encuesta

Queremos validar que una función obtener_precio() devuelva un valor correcto.

¿Cómo arreglar el código?

```
precio = obtener_precio()
assert precio   # Pasa si precio es truthy
```

A

```
precio = obtener_precio()
assert precio is not None
```

B

```
precio = obtener_precio()
assert isinstance(int, precio)
```

C

```
precio = obtener_precio()
assert precio >= 0
```

Encuesta

Queremos validar que una función obtener_precio() devuelva un valor correcto.

¿Cómo arreglar el código?

```
precio = obtener_precio()
assert precio   # Pasa si precio es truthy
```

A

```
precio = obtener_precio()
assert precio is not None
```

B

```
precio = obtener_precio()
assert isinstance(int, precio)
```

C

```
precio = obtener_precio()
assert precio >= 0
```

Testing en el contexto de datos

Hasta ahora hemos hablado de testing enfocado en **código**: verificar que funciones y módulos hagan lo que se espera, bajo distintos escenarios.

Cuando trabajamos con **datos**, la lógica es la misma: el código que procesa datos también debe ser **testeado** para garantizar su comportamiento correcto.

Ejemplo:

Queremos validar que una **fórmula estadística** implementada en nuestro código calcule correctamente los resultados sobre un conjunto de datos.

Más allá del código: calidad de los datos

En proyectos de **data engineering** o **data science**, además de verificar el código, es fundamental asegurar la **calidad de los datos.**

- ¿Existen valores faltantes o inconsistentes?
- ¿Se respetan los tipos de datos esperados en cada columna?
- ¿Existen duplicados que no deberían estar?
- ¿Los datos cumplen con las reglas del negocio (ejemplo: fechas no pueden ser futuras, IDs deben ser únicos)?

Más allá del código: calidad de los datos

El testing de código sigue siendo esencial, pero en entornos basados en datos añadimos un nuevo eje: el **testing de calidad de datos.**

Como veremos en la próxima clase, la importancia de la **validación de datos** se vuelve aún más evidente a medida que crece la cantidad de información, ya que los errores escalan y pueden tener un impacto mucho mayor.

Resumen

El **testing** es esencial para asegurar que el código cumpla con los requisitos y evitar errores costosos.

Existen distintos tipos de pruebas (**unitarias, de integración, end-to-end**), cada una con su alcance y costo.

En proyectos basados en datos, además de probar el código, será clave validar la **calidad de los datos.**

INTRODUCCIÓN AL ANÁLISIS DE DATOS

Datos

Los datos son cualquier clase de información que identifica conceptos, números o relaciones entre situaciones naturales del mundo que nos rodea.

Nombres de personas, estadísticas de rendimiento de un computador, información económica de un país, etc.

Datos

Los datos son información: nombres, estadísticas, registros, transacciones, etc.

Se originan en ciencia, comercio, medicina, redes sociales, entretenimiento, entre otros contextos.

Toda información capturada puede considerarse un **dato**, que a su vez permite generar nueva información mediante operaciones estadísticas, reportes o visualizaciones.

Ejemplos

Colisionador de Hadrones (Suiza)

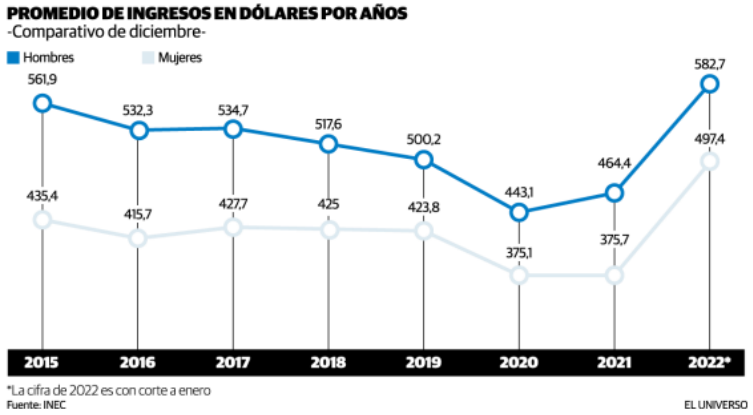
•Proyecto de investigación sobre la naturaleza de las partículas.

•Durante los experimentos se generan más de **40 terabytes de datos por segundo**.

Ejemplos

INEC (Ecuador)

- Durante un **censo**, el INEC captura información de **millones de ecuatorianos.**
- Además, produce reportes periódicos con datos **micro y macroeconómicos.**



Ciclo de vida

Los datos son materia prima para transformar información en conocimiento.

¿Cuál es el proceso general para **obtener nueva información** mediante el **análisis de datos**?

Ciclo de vida

Siendo analista

Un **analista de datos** trabaja en las distintas etapas del ciclo de análisis.

Antes de las computadoras, este trabajo se hacía manualmente para obtener estimaciones.

Hoy, las computadoras permiten realizar el mismo proceso de forma **más rápida, precisa y eficiente**.

Siendo analista

De hecho, no siempre es necesario programar: herramientas como **Excel** pueden actuar como base de datos y realizar operaciones matemáticas.

Sin embargo, en **contextos más complejos** estas soluciones se vuelven limitadas, y es necesario recurrir a **Python** u otros lenguajes especializados para manejar y analizar datos de manera eficiente.

¿Por qué usar Python?

Su ecosistema de **librerías** potencia al analista con herramientas para leer archivos, conectarse a bases de datos y trabajar en múltiples entornos.

Ciclo de vida

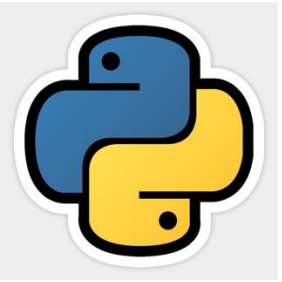
Fuentes de Datos

Obtener -> Limpiar -> Integrar -> Analizar -> Visualizar

Reportes, exploración, etc.

En Python esto simplemente identifica los diferentes mecanismos disponibles para acceder a datos, ya sea por medio de archivos, bases de datos, o servicios web.

Ciclo de vida



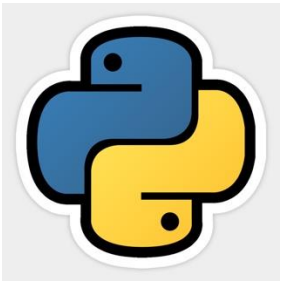
Fuentes de Datos

Obtener -> **Limpiar** -> Integrar -> Analizar -> Visualizar

Reportes, exploración, etc.

Los datos vienen en muchas formas, a veces estructurados, a veces no. Limpiar se refiere al proceso de estandarizarlos en forma que puedan ser utilizados para el análisis correspondiente.

Ciclo de vida



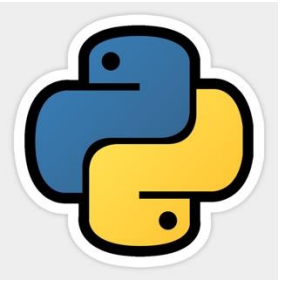
Fuentes de Datos

Obtener -> Limpiar -> Integrar -> Analizar -> Visualizar

Integrar simplemente hace referencia a unificar las fuentes de datos requeridas en el contexto analítico.

Reportes, exploración, etc.

Ciclo de vida



Fuentes de Datos

Obtener -> Limpiar -> Integrar -> Analizar -> Visualizar

Reportes, exploración, etc.

Analizar hace referencia a la información que queremos obtener de esos datos, esto depende mucho del problema en el que estamos enfocándonos.

Ciclo de vida

Fuentes de Datos

Obtener -> Limpiar -> Integrar -> Analizar -> Visualizar

Reportes, exploración, etc.

Visualizar hace referencia a las distintas formas en las que podemos explorar nuestro análisis en el contexto actual, por ejemplo, creando gráficos, o imprimiendo tablas numéricas.

Ciclo de vida

Fuentes de Datos

Obtener -> Limpiar -> Integrar -> Analizar -> Visualizar

Reportes, exploración, etc.

El objetivo final del análisis es siempre contestar una o más preguntas. La forma en que esto se haga depende del problema y la razón por la que lo ejecutamos.

Análisis en Python

Obtener → leer datos desde archivos, bases de datos o APIs.
Librerías: `pandas`, `sqlalchemy`, `requests`, `duckdb`.

Limpiar → corregir tipos, valores nulos y errores.
Librerías: `numpy`, `pydantic`, `pandera`

Integrar → unir y combinar distintas fuentes.
Librerías: `duckdb`, `ibis`.

Analizar → aplicar estadísticas, modelos y métricas.
Librerías: `pandas`, `numpy`, `scipy`, `statsmodels`, `scikit-learn`.

Visualizar → comunicar hallazgos con gráficos o reportes.
Librerías: `matplotlib`, `seaborn`, `plotly`.

Análisis

El **análisis de datos** es el proceso de obtener **nueva información** a partir de métodos analíticos.

Puede ir desde lo más simple hasta lo más complejo:

- **Estadística básica** → promedios, medianas.
- **Modelos avanzados** → regresiones, inferencias, predicciones.
- **Análisis visual** → responder preguntas explorando gráficas o tablas de datos.

Procesamiento de datos

El primer paso en el análisis de datos

Para ello debemos responder dos preguntas clave:

- **¿Dónde está la información?** (archivos, bases de datos, servicios web, etc.)
- **¿Cómo debo ajustarla para cumplir mis objetivos?** (limpieza, transformación, estandarización).

Almacenamiento

Los **datos** pueden almacenarse de distintas formas, y según su tipo se requieren procesos diferentes para **extraerlos y prepararlos** antes del análisis.

Las formas más comunes son: **archivos de datos**, **bases de datos** y **servicios web**.

Bases de datos

Una **base de datos** es una colección organizada de información estructurada, almacenada electrónicamente y gestionada por un **sistema de gestión de bases de datos (DBMS)**.

Relacionales:

Los datos se organizan en **tablas con filas y columnas** que pueden relacionarse entre sí. Ejemplos: **MySQL, PostgreSQL, SQLite**.

No relacionales (NoSQL):

Almacenan la información de otras formas: objetos (MongoDB, Cassandra) o en grafos (Neo4j).

Bases de datos

Las **bases de datos relacionales** se gestionan principalmente mediante **SQL** (*Structured Query Language*).

Este lenguaje permite a usuarios y aplicaciones **acceder y manipular la información** de forma estructurada y eficiente.

Bases de datos

Las **bases de datos relacionales** se gestionan principalmente mediante **SQL** (*Structured Query Language*).

Este lenguaje permite a usuarios y aplicaciones **acceder y manipular la información** de forma estructurada y eficiente.

En esta clase cubriremos cómo conectarse a **bases de datos relacionales** usando Python, aunque también existen librerías bien documentadas para trabajar con **bases de datos NoSQL.**

Sqlite3 en Python

SQLite3 es una base de datos relacional muy sencilla que no requiere un motor o servidor, ya que los datos se almacenan directamente en uno o más archivos. Python incluye soporte para SQLite mediante la librería estándar **sqlite3**, disponible de forma predeterminada.

```python
import sqlite3
from sqlite3 import Error

def crear_conexion(parametros):
    conexion = None
    try:
        conexion = sqlite3.connect(parametros)
        print("Conexión exitosa")
    except Error as e:
        print(f"Error '{e}'")

    return conexion

conexion =
crear_conexion("ubicacion/del/archivo.sqlite")
conexion.execute("select * from tabla")
resultados = conexion.fetchall()
print(resultados)
conexion.close()
```

https://docs.python.org/3/library/sqlite3.html

MySQL en Python

A diferencia de **SQLite3**, Python no incluye de forma nativa módulos para interactuar con **MySQL**, pero existen muchos disponibles en el repositorio público.

```python
import mysql.connector
from mysql.connector import Error

def crear_conexion(servidor, usuario, clave):
    conexion = None
    try:
        conexion = mysql.connector.connect(
            host=servidor,
            user=usuario,
            passwd=clave
        )
        print("Conexión a MYSQL existosa")
    except Error as e:
        print(f"Error '{e}'")
    return conexion

conexion = crear_conexion("localhost", "miusuario", "")
cursor = conexion.cursor()
cursor.execute("SELECT * FROM tabla")
resultado = cursor.fetchall()
conexion.close()
```

```
python -m pip install mysql-connector-python
```

https://dev.mysql.com/doc/connector-python/en/

Otras bases de datos

¡En realidad, existen librerías para cualquier sistema de base de datos común!

- PostreSQL
- Cassandra
- MongoDB
- etc, etc, etc.

Siempre revisar la documentacion official.

Escribiendo SQL en Python

Escribir SQL directamente en Python conlleva varios riesgos, principalmente relacionados con la **complejidad** del código y la **seguridad**.

Encuesta

¿Cuál es el principal riesgo en este ejemplo?

```
from flask import Flask, request
from sqlalchemy import create_engine, text

app = Flask(__name__)
engine = create_engine('sqlite:///mi_base_de_datos.db')

@app.route('/buscar')
def buscar():
  # variable externa provista por un usuario
   nombre = request.args.get('nombre')
   with engine.connect() as conn:
      consulta = "SELECT * FROM usuarios WHERE nombre = " + nombre
      resultados = conn.execute(consulta).fetchall()
   return str(resultados)

if __name__ == '__main__':
   app.run()
```

Encuesta

¿Cuál es el principal riesgo en este ejemplo?

Inyección SQL, riesgo de seguridad

El valor:

```
"; DELETE * FROM usuarios;
```

Genera:

```
SELECT * FROM usuarios WHERE nombre = "";
DELETE * FROM usuarios;
```

```python
from flask import Flask, request
from sqlalchemy import create_engine, text

app = Flask(__name__)
engine = create_engine('sqlite:///mi_base_de_datos.db')

@app.route('/buscar')
def buscar():
  # variable externa provista por un usuario
   nombre = request.args.get('nombre')
   with engine.connect() as conn:
      consulta = "SELECT * FROM usuarios WHERE nombre = " + nombre
      resultados = conn.execute(consulta).fetchall()
   return str(resultados)

if __name__ == '__main__':
   app.run()
```

Declaraciones preparadas

Permiten ejecutar consultas SQL de forma **más eficiente y segura** al precompilar la consulta y reutilizarla con distintos parámetros.

- Previenen ataques de **inyección SQL**.
- Mejoran el **rendimiento**, ya que la base de datos ejecuta una consulta ya compilada.
- Facilitan la reutilización y mantenimiento del código.

Declaraciones preparadas

```
from sqlalchemy import create_engine, text

# Conexión a la base de datos
engine = create_engine('sqlite:///mi_base_de_datos.db')
connection = engine.connect()

# Ejemplo de declaración preparada
stmt = text("SELECT * FROM usuarios WHERE nombre = :nombre")
result = connection.execute(stmt, {"nombre": "Juan"})

for row in result:
    print(row)
```

sqlalchemy

SQLAlchemy ofrece flexibilidad para trabajar con bases de datos relacionales de manera más eficiente, especialmente en aplicaciones complejas.

Características principales:

•Modelo relacional que permite operar usando **objetos nativos de Python**.

•Sistema eficiente para manejar **objetos y operaciones relacionales** sin escribir SQL directamente.

•Herramientas para **generar, explorar e introspeccionar** bases de datos.

https://pypi.org/project/SQLAlchemy/

sqlalchemy

Interfaz consistente usando clases y objetos en lugar de SQL

```
from sqlalchemy import create_engine, Column, Integer, String, MetaData, Table

from sqlalchemy import create_engine
engine = create_engine('sqlite:///mi_base_de_datos.db')
connection = engine.connect()

metadata = MetaData()
usuarios = Table('usuarios', metadata,
    Column('id', Integer, primary_key=True),
    Column('nombre', String),
    Column('edad', Integer)
)

ins = usuarios.insert().values(nombre='Juan', edad=30)
connection.execute(ins)
```

https://pypi.org/project/SQLAlchemy/

sqlalchemy

- Se integra con muchos frameworks de servicios web como **Flask** y **Django**.
- Permite prevenir errores comunes y trabajar con un **lenguaje unificado** dentro de aplicaciones mejor estructuradas.
- Aunque suele usarse más en **aplicaciones** que en análisis de datos, es útil conocerlo en escenarios donde se requiere manejar bases de datos de forma robusta.

SQL para análisis

Además de usarse en aplicaciones, **SQL** también puede ser una herramienta poderosa para el **análisis de datos**.

Con herramientas como **DuckDB** o **Trino**, es posible:

- Ejecutar consultas SQL directamente desde Python.
- Analizar archivos locales (CSV, Parquet) de forma interactiva.
- Combinar el poder de SQL con librerías como **pandas**.

En las próximas clases veremos cómo aprovechar estas herramientas.

Archivos

Los **archivos** son una de las formas más comunes de almacenamiento de información

- Son clave en análisis de datos porque pueden transferirse fácilmente entre sistemas y proyectos.
- Funcionan como **fuentes primarias** en muchos pipelines analíticos.

Formatos

CSV (Comma-Separated Values)

- Representa datos tabulares con separadores (, o ;).
- Muy usado por su simplicidad y compatibilidad (Excel, Python, R, etc.).

Parquet

- Formato tabular **columna-orientado**, optimizado para grandes volúmenes.
- Ideal para análisis en pipelines modernos (ej. con **DuckDB**, **pandas**, Spark).

Existen muchos más formatos de archivos de datos, pero en este curso nos enfocaremos en **CSV** y **Parquet**, ya que son los más comunes y prácticos en contextos de análisis.

Archivos CSV

CSV – Comma Separated Values
Formato **tabular** donde cada fila representa un registro y las columnas se separan por comas (,).

Es un formato de **texto plano**, lo que lo hace simple y ampliamente compatible. Puede abrirse y manipularse fácilmente con múltiples herramientas, incluyendo **Excel**, editores de texto y librerías de programación como **pandas** en Python.

```
Usuario;Identificador;Nombre;Apellido
booker12;9012;Rachel;Booker
grey07;2070;Laura;Grey
johnson81;4081;Craig;Johnson
jenkins46;9346;Mary;Jenkins
smith79;5079;Jamie;Smith
```

Archivos CSV

Al ser un archivo de texto plano, es bastante fácil leer este archivo como cualquier otro:

```
with open("ubicacion.csv") as archivo:
    for fila in archivo:
        # Dividiamos las columnas usando la , pero tambien puede ser ;
        columnas = fila.split(",")
        print(columnas)
```

Es mejor usar la **librería estándar csv de Python**, que gestiona detalles importantes de forma automatizada.

```
import csv
with open("ubicacion.csv") as archivo:
    # Definir el delimitador que separa las columnas
    filas = csv.reader(archivo, delimiter=';')
    for fila in filas:
        # Automáticamente se convirtieron en columnas
        print(fila[0])

# Podemos fácilmente agregar filas!
with open('ubicacion.csv', 'w') as archivo:
    escritor = csv.writer(archivo, delimiter=';')
    escritor.writerow(['Pollo', 'Papas'])
```

Archivos parquet

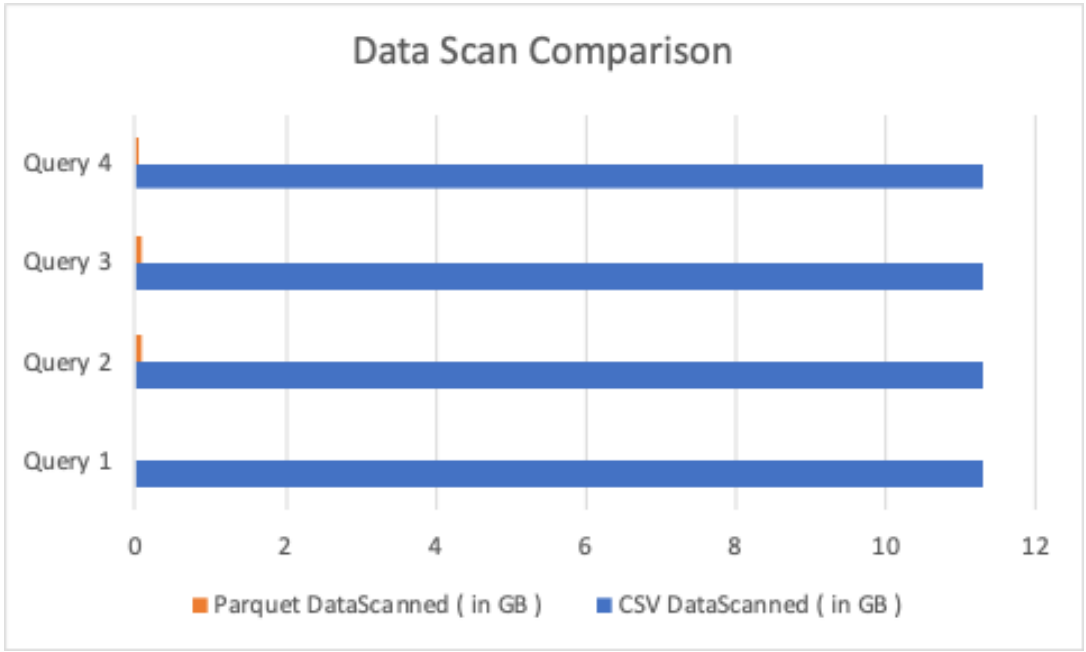
- Formato de almacenamiento binario, **columnar**.
- Diseñado para manejar grandes volúmenes de datos de forma eficiente.
- Común en entornos de **big data** y análisis distribuido.

Archivos parquet

CSV → simple, universal, excelente para datos pequeños o intercambio rápido.

Parquet → eficiente y escalable, mejor para análisis y pipelines complejos.

Archivos parquet vs csv



Archivos parquet

Python incluye soporte para Parquet a través de librerías externas.

Las más comunes son:

- **pyarrow** → biblioteca de Apache Arrow, muy eficiente.
- **fastparquet** → implementación ligera, integrada con pandas.

Archivos y procesos analíticos

Con **pandas** podemos leer y escribir datos en formato Parquet y CSV como DataFrames.

Con **DuckDB** podemos consultar archivos Parquet o CSV directamente con **SQL**.

Ambos pueden ser integrados datos de forma y eficiente en pipelines más complejas.

Los archivos son clave al construir **pipelines orquestados**, donde se combinan diferentes fuentes y formatos de datos.

Laboratorio

El Github incluye el segundo laboratorio para repasar los conceptos.

https://github.com/danoc93/ista-python-analisis-2025

Aviso:

- Propuesta del proyecto ahora disponible