Manejo de Errores y Logging en ETL - Día 5

Change Status

in-progress 40 min

Learning Objectives

1. Implementar manejo de excepciones robusto en pipelines ETL usando try/except
2. Configurar logging estructurado para trazabilidad y debugging
3. Diseñar estrategias de recuperación automática ante fallos

Actividades y Aprendizajes

Aprende todo sobre funciones y módulos en Python con ejemplos prácticos.

Task 1: Manejo de Excepciones en Python (10 minutos)

Las excepciones representan los **momentos de crisis** en un pipeline ETL, donde algo sale mal y el sistema debe decidir cómo responder. Un buen manejo de excepciones transforma fallos potenciales en **oportunidades de recuperación** o al menos **terminación graceful**.

Try/Except: La Base del Manejo de Errores

Estructura fundamental: Código riesgoso protegido por bloques de manejo.

```
try:
    # Código que puede fallar
    df = pd.read_csv('datos_posiblemente_corruptos.csv')
    resultado = funcion_que_puede_fallar(df)
except FileNotFoundError:
    # Manejar archivo inexistente
    print("Archivo de datos no encontrado")
```

```
    df = pd.DataFrame()  # DataFrame vacío como fallback
except pd.errors.EmptyDataError:
    # Manejar archivo vacío
    print("Archivo CSV está vacío")
    df = pd.DataFrame()
except Exception as e:
    # Manejar cualquier otro error
    print(f"Error inesperado: {e}")
    raise  # Re-lanzar si no podemos manejar
```

Tipos de excepciones comunes en ETL:

`FileNotFoundError`: Archivos inexistentes
`pd.errors.ParserError`: CSV mal formateado
`sqlite3.OperationalError`: Problemas de base de datos
`requests.exceptions.RequestException`: Fallos de API
`ValueError`: Datos inválidos en transformaciones
Estrategias de Recuperación

Fallback values: Proporcionar valores por defecto cuando falla una operación.

```
def cargar_datos_con_fallback(ruta_archivo, ruta_backup=None):
    try:
        return pd.read_csv(ruta_archivo)
    except FileNotFoundError:
        if ruta_backup:
            print(f"Archivo principal no encontrado, intentando backup: {ruta_backup}")
            return pd.read_csv(ruta_backup)
        else:
            print("Ni archivo principal ni backup disponibles")
            return pd.DataFrame()  # DataFrame vacío
```

Reintentos automáticos: Para operaciones inestables como conexiones de red.

```
import time

def operacion_con_reintento(funcion, max_reintentos=3, delay=1):
    for intento in range(max_reintentos):
        try:
            return funcion()
        except Exception as e:
```

```
            if intento == max_reintentos - 1:
                raise e  # Último intento fallido
            print(f"Intento {intento + 1} falló: {e}. Reintentando en {delay}s...")
            time.sleep(delay)
            delay *= 2  # Exponential backoff
```

Task 2: Logging Estructurado para ETL (10 minutos)

El logging transforma el **debugging reactivo** en **monitoreo proactivo**, permitiendo entender qué sucede en el pipeline incluso cuando todo funciona correctamente.

Configuración Básica de Logging

Niveles de severidad: DEBUG < INFO < WARNING < ERROR < CRITICAL

```
import logging

# Configurar logging básico
logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('etl_pipeline.log'),  # Archivo
        logging.StreamHandler()  # Consola
    ]
)

logger = logging.getLogger(__name__)
```

Uso en pipeline ETL:

```
def extract_data(ruta_archivo):
    logger.info(f"Iniciando extracción desde: {ruta_archivo}")
    try:
        df = pd.read_csv(ruta_archivo)
        logger.info(f"Extracción exitosa: {len(df)} registros cargados")
        return df
    except Exception as e:
        logger.error(f"Error en extracción: {e}")
        raise
```

```
def transform_data(df):
    logger.info("Iniciando transformación de datos")
    registros_originales = len(df)

    # Transformaciones
    df = df.dropna()
    df['total'] = df['cantidad'] * df['precio']

    logger.info(f"Transformación completada: {registros_originales} -> {len(df)} registros")
    return df
```

Logging Context-Aware

Información contextual: Incluir detalles relevantes en cada mensaje.

```
def load_data(df, tabla_destino, conn):
    logger.info(f"Iniciando carga a tabla: {tabla_destino}")
    logger.info(f"Registros a cargar: {len(df)}")

    try:
        df.to_sql(tabla_destino, conn, index=False, if_exists='append')
        logger.info(f"Carga exitosa a {tabla_destino}")
    except Exception as e:
        logger.error(f"Error cargando a {tabla_destino}: {e}")
        # Intentar rollback si es transacción
        conn.rollback()
        raise
```

Logging Jerárquico para Pipelines Complejos

Separar concerns: Diferentes loggers para diferentes componentes.

```
# Loggers específicos por módulo
extract_logger = logging.getLogger('etl.extract')
transform_logger = logging.getLogger('etl.transform')
load_logger = logging.getLogger('etl.load')

# Configurar diferentes niveles
extract_logger.setLevel(logging.DEBUG)
load_logger.setLevel(logging.ERROR)  # Solo errores en carga
```

Task 3: Arquitectura de Pipeline Robusto (10 minutos)

Un pipeline ETL robusto combina manejo de errores, logging, y estrategias de recuperación para crear un **sistema confiable** que puede manejar fallos gracefully.

Patrón de Pipeline con Manejo Integral

Estructura modular: Cada fase es independiente pero coordinada.

```
class ETLPipeline:
    def __init__(self):
        self.logger = logging.getLogger('etl.pipeline')

    def run_pipeline(self):
        self.logger.info("=== INICIANDO PIPELINE ETL ===")

        try:
            # Fase 1: Extracción
            data = self.extract()

            # Fase 2: Transformación
            transformed_data = self.transform(data)

            # Fase 3: Carga
            self.load(transformed_data)

            self.logger.info("=== PIPELINE ETL COMPLETADO EXITOSAMENTE ===")

        except Exception as e:
            self.logger.error(f"=== PIPELINE ETL FALLÓ: {e} ===")
            self.handle_pipeline_failure(e)
            raise

    def extract(self):
        # Lógica de extracción con manejo de errores
        pass

    def transform(self, data):
        # Lógica de transformación
        pass

    def load(self, data):
        # Lógica de carga
        pass

    def handle_pipeline_failure(self, error):
        # Lógica de recuperación o cleanup
```

```
        self.logger.info("Ejecutando cleanup post-falló")
        # Enviar alertas, rollback, etc.
```

Estrategias de Recupería

Puntos de checkpoint: Guardar estado intermedio para reanudar desde último punto exitoso.

```
class CheckpointETL(ETLPipeline):
    def __init__(self):
        super().__init__()
        self.checkpoint_file = 'etl_checkpoint.pkl'

    def save_checkpoint(self, data, phase):
        import pickle
        checkpoint = {'data': data, 'phase': phase, 'timestamp': pd.Timestamp.now()}
        with open(self.checkpoint_file, 'wb') as f:
            pickle.dump(checkpoint, f)
        self.logger.info(f"Checkpoint guardado en fase: {phase}")

    def load_checkpoint(self):
        try:
            with open(self.checkpoint_file, 'rb') as f:
                return pickle.load(f)
        except FileNotFoundError:
            return None

    def run_with_checkpoint(self):
        checkpoint = self.load_checkpoint()
        if checkpoint:
            self.logger.info(f"Reanudando desde checkpoint: {checkpoint['phase']}")
            # Lógica para reanudar desde checkpoint
        else:
            self.run_pipeline()
```

Monitoreo y Alertas

Métricas de éxito: Rastrear completion rate, tiempos de ejecución, volumen de datos.

```
class MonitoredETL(ETLPipeline):
    def __init__(self):
        super().__init__()
        self.metrics = {
            'start_time': None,
            'end_time': None,
```

```
            'records_processed': 0,
            'errors_count': 0
        }

    def run_with_monitoring(self):
        self.metrics['start_time'] = pd.Timestamp.now()
        try:
            self.run_pipeline()
            self.metrics['end_time'] = pd.Timestamp.now()
            self.report_success()
        except Exception as e:
            self.metrics['end_time'] = pd.Timestamp.now()
            self.metrics['errors_count'] += 1
            self.report_failure(e)
            raise

    def report_success(self):
        duration = self.metrics['end_time'] - self.metrics['start_time']
        self.logger.info(f"Pipeline exitoso - Duración: {duration}")
        # Aquí se podrían enviar alertas de éxito

    def report_failure(self):
        duration = self.metrics['end_time'] - self.metrics['start_time']
        self.logger.error(f"Pipeline falló - Duración: {duration} - Errores: {self.metrics['errors_count']}")
        # Aquí se enviarían alertas de fallo
```