Documentación Técnica:

PROYECTO 1, 2 Y 3

Contenido

Introducción

Este documento describe un sistema integral desarrollado para el análisis y medición de código Python, que abarca tres proyectos interconectados y complementarios: Proyecto 1: Calculo de Líneas Lógicas y Físicas, Proyecto 2: Análisis y Conteo de Líneas de Código por Estructura y Clase y Proyecto 3: Comparación de Cambios Entre Versiones.

En el Proyecto 1, se implementó un sistema de análisis de código que se centra en el conteo de líneas lógicas (LLOC) y su categorización, tales como definiciones de funciones, clases, estructuras de control, y otras construcciones esenciales de Python. Este proyecto también establece un enfoque para la gestión de errores comunes en el código, asegurando que se cumpla con las buenas prácticas de programación.

El Proyecto 2 se construye sobre el análisis realizado en el Proyecto 1, pero con un enfoque más orientado a la organización del código. En este proyecto, se analiza y cuenta las líneas de código de un archivo Python, diferenciando las líneas totales, las líneas de cada clase y las líneas de cada método o función. En el caso de la programación orientada a objetos (OO), se cuenta el número de líneas por clase y método, mientras que en programación estructurada se cuentan las líneas de código de cada función o procedimiento. Este proyecto reutiliza el código del Proyecto 1, adaptándolo para proporcionar un análisis más detallado sobre la estructura del código.

El Proyecto 3 lleva el sistema un paso más allá, permitiendo la comparación entre diferentes versiones de un archivo Python. Este proyecto reutiliza el código implementado en el Proyecto 2, adaptando las funciones existentes para permitir una comparación eficaz entre versiones, con el análisis de líneas modificadas, añadidas o eliminadas. Además, se incorpora la capacidad de manejar excepciones relacionadas con el análisis de diferencias y el manejo de código formateado incorrectamente.

Tecnologías Utilizadas

1. Lenguaje de Programación: Python 3.12
2. Entorno de Desarrollo: Visual Studio Code
3. Análisis de Código:
    - pylint: Para verificar estándares de codificación como longitud máxima de líneas y profundidad de anidamiento.
    - pytest: Para pruebas automatizadas.

4. Control de Versiones:
    - GitHub: Control de cambios, manejo de issues y proyectos.
    - GitHub Actions: Automatización de pruebas con pylint.
5. Comunicación y Colaboración: Discord.

Características del Proyecto 1:

Contar Líneas Físicas y Lógicas

Funcionamiento General del Sistema

Entradas

1. Ubicación del archivo Python a analizar.
2. Parámetros adicionales:
    - Si el usuario desea imprimir o no las métricas en formato de tabla.

Salidas

1. Archivo metricas.json con las métricas calculadas:
    - Líneas físicas (LF).
    - Líneas lógicas (LL).
2. Tabla opcional mostrando las métricas en la terminal (si el usuario lo solicita).

Errores Comunes

- Archivo no encontrado: Si el archivo proporcionado no existe o no es accesible.
- Archivo inválido: Si el archivo no es un archivo Python o no cumple con los estándares mínimos.
- Estructuras no soportadas: Si el código contiene estructuras que no pueden ser procesadas.

Qué hace el sistema:

- Cuenta líneas físicas: Cualquier línea de código Python que no sea un comentario o línea en blanco.
- Cuenta líneas lógicas: Líneas que representan estructuras clave como if, for, def, etc.
- Estructura jerárquica: Representa el código como un árbol donde los nodos hijos representan anidamientos dentro de estructuras como funciones o ciclos.
- Verifica estándares: Evalúa si el archivo cumple con los estándares de codificación para estructuras como ternarios, lambdas y comprehensions.

Qué no hace el sistema:

- No identifica si el código es compilable.
- No verifica estándares de nombrado o calidad de comentarios/docstrings.
- No ordena ni valida importaciones.
- No muestra específicamente cuáles son las líneas físicas y lógicas identificadas.

Verificación de Estándares de Codificación

Se verifica que el archivo cumpla con los estándares básicos para estructuras como:

- Ternarios: Evaluación de condiciones en una sola línea.
- Lambdas: Uso correcto de funciones anónimas.
- Comprehensions: Generación de listas, diccionarios o conjuntos con expresiones compactas.

Conteo de Líneas Físicas y Lógicas

Líneas físicas (LF):

- Contabiliza todas las líneas que contienen código.
- Se excluyen comentarios y líneas en blanco.

Líneas lógicas (LL):

- Estructuras como if, while, def, class, y otras instrucciones clave cuentan como una línea lógica.

Anidamiento:

- Se considera la profundidad del código, contabilizando correctamente las líneas anidadas.

Almacenamiento en JSON

1. Formato del archivo:
   Las métricas de las pruebas se almacenan en un archivo llamado contador_lineas_registro.json, con la siguiente estructura:


```
{
  "test_1.py": {
    "nombre_archivo": "test_1.py",
    "lineas_logicas": 35,
    "lineas_fisicas": 92
  },
  "test_2.py": {
    "nombre_archivo": "test_2.py",
    "lineas_logicas": 8,
    "lineas_fisicas": 19
  }
}
```


Propósito:

- Facilita la persistencia de resultados para análisis posteriores.
- Permite visualizar el resultado de los cálculos de las líneas físicas y lógicas después de realizar el proceso o testeo.

5.2 Descripción de Módulos y Relaciones

5.2.1 Módulo Principal

main.py

- Punto de entrada para la aplicación.
- Maneja los argumentos de línea de comandos.
- Orquesta el flujo entre módulos.
- Dependencias: core.lector_archivo, core.contador_logico, core.almacenamiento_metricas.

5.2.2 Módulos Core

core/lector_archivo.py

- Responsable de leer archivos Python.
- Validación básica de archivos.
- Retorna el contenido del archivo para su procesamiento.
- Dependencias: utils.validador.

core/contador_logico.py

- Implementa reglas lógicas para el conteo de LOC (líneas de código lógico).
- Procesa el contenido del archivo línea por línea.
- Retorna un objeto de métricas.
- Dependencias: models.metricas.

core/almacenamiento_metricas.py

- Administra el almacenamiento y recuperación en formato JSON.

- Crea/actualiza el historial de métricas.
- Dependencias: models.metricas.

5.2.3 Modelos

models/metricas.py

- Estructuras de datos para métricas.
- Objetos inmutables para los resultados.
- Sin dependencias.

5.2.4 Utilidades

utils/validador.py

- Funciones de validación de entradas.
- Verificación de extensiones de archivo.
- Validación de rutas.
- Sin dependencias.

utils/formatters.py

- Utilidades para formateo de salida.
- Generación de tablas para resultados.
- Sin dependencias.

Automatización e Integración Continua

- GitHub Actions:
    - Automatiza pruebas con pylint.
    - Rechaza cambios que no cumplan con los estándares.
- Uso de Variables de Entorno:

  C:\Users\<TU_USUARIO>\AppData\Local\Programs\Python\PythonXX\Scripts\ en la variable de entorno PATH para ejecutar el proyecto.

Características del Proyecto 2:

Contar Líneas Totales de un Programa

Flujo de Trabajo del Sistema

Entrada

1. El usuario ingresa la ubicación del archivo Python que desea analizar.
2. El usuario también tiene la opción de decidir si quiere que el programa imprima las métricas detalladas en forma de tabla (líneas físicas, lógicas, etc.).

Procesamiento

1. El programa se inicia con el módulo principal (__main__.py).
2. El analizador se encarga de procesar el archivo Python ingresado:
    - Se verifica que el archivo sea un archivo Python válido y accesible.
    - Se genera un árbol sintáctico que representa la estructura del código Python.
    - Cada clase, método, y función/procedimiento se identifica y se cuentan las líneas correspondientes.
    - El programa reutiliza las funcionalidades del Proyecto 1 para contar las líneas físicas y lógicas, asegurándose de que se siga el estándar de codificación para cada clase, método y función.

Análisis de Clases y Métodos

1. Se identifican las clases dentro del código y se contabilizan las líneas de código de cada una.
2. Dentro de cada clase, se cuentan los métodos, registrando las líneas correspondientes.
3. Si el código utiliza programación estructurada, las funciones son analizadas y sus líneas de código son contabilizadas.

   ### Verificación de Estándar de Codificación

   El programa verifica que se sigan los estándares de codificación en cuanto a las estructuras lógicas y de control (como ternarios, lambdas, y comprehensions) dentro de las clases y funciones.

Salida

1. Las métricas de líneas de código físicas y lógicas, por clase, por método y por función, se almacenan en un archivo metricas.json.
2. Si el usuario lo solicita, las métricas se imprimen en una tabla que muestra:
   - Líneas físicas y lógicas del archivo.
   - Líneas de código por clase, método y función.
   - Número total de métodos o funciones en el archivo.

Errores y Manejo

El programa manejará los siguientes posibles errores:

- Si el archivo no existe o no es accesible.
- Si el archivo no es un archivo Python.
- Si se encuentran violaciones al estándar de codificación (por ejemplo, funciones demasiado largas, sin docstrings, etc.).

Componentes del Proyecto

1. Reutilización del Proyecto 1

El Proyecto 2 reutiliza los siguientes componentes del Proyecto 1:

- Contadores de líneas físicas y lógicas.
- Análisis de estructuras lógicas (if, while, ternarios, etc.).
- Verificación de estándares de codificación.

Nuevo Análisis y Contabilización de Clases, Métodos y Funciones

A diferencia del Proyecto 1, en este proyecto se incorpora la identificación de clases y métodos, así como el conteo de las líneas de código asociadas a cada uno:

- Clases: Se identifican utilizando el patrón class ClassName.
- Métodos: Se identifican dentro de las clases utilizando el patrón def method_name.
- Funciones: Se identifican por el patrón def function_name fuera de las clases.

2. Recuperación de Información

El archivo metricas.json se puede consultar y analizar en cualquier momento para obtener un desglose detallado de las métricas. Si el usuario ha solicitado imprimir las métricas, estas se mostrarán en forma de tabla utilizando el módulo formateador_metricas.py.

Características y Limitaciones del Sistema

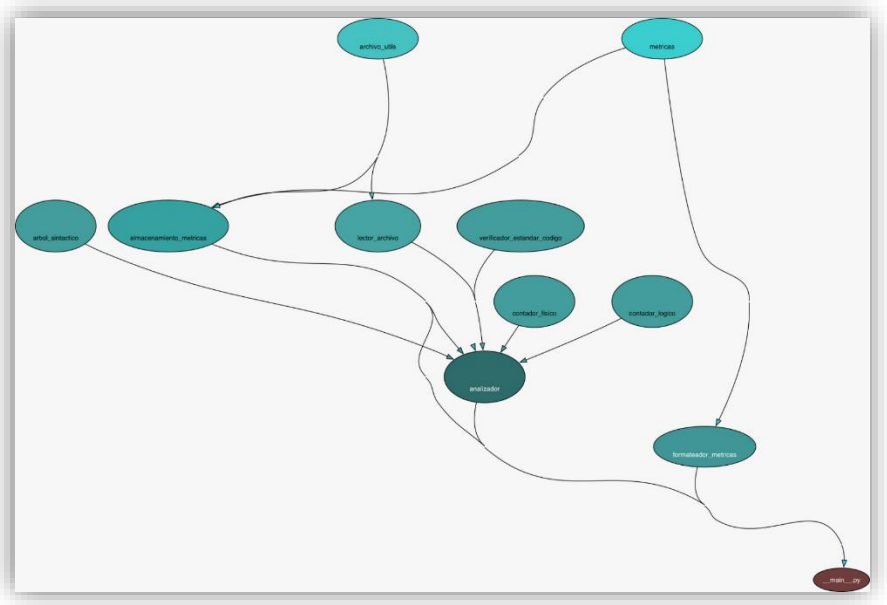
Lo que sí hace el sistema:

1. Contabiliza las líneas de código totales, físicas y lógicas.
2. Cuenta las líneas de código por clase y por método dentro de las clases.
3. Cuenta las líneas de código de funciones en programas estructurados.
4. Verifica que el código siga el estándar de codificación para estructuras lógicas.
5. Genera métricas y las almacena en un archivo metricas.json.
6. Imprime las métricas en formato de tabla si es solicitado por el usuario.

Lo que no hace el sistema:

1. No verifica si el código es compilable o si existen errores de sintaxis.
2. No comprueba el estándar de nombrado de clases, métodos y funciones.
3. No valida si el código tiene comentarios significativos o docstrings.
4. No asegura que las importaciones estén en el orden correcto.

Diagrama de Secuencia



Características del Proyecto 3:

Contar Número de Cambios entre Versiones

Estructura General del Sistema:

1. Entradas:
    - Rutas de archivos Python para analizar.
    - Ruta para almacenar las métricas.
    - Opciones de línea de comandos para mostrar tablas de métricas y cambios.
2. Salidas:
    - Métricas de líneas de código físicas y lógicas.
    - Comparación de cambios entre versiones de archivos.
    - Archivos Python con comentarios añadidos (indicando líneas borradas o añadidas).
    - Tablas de métricas y cambios mostradas en la consola.

Comparación de Cambios entre Archivos:

- Determinación de Cambios:
    - El sistema utiliza nodos de un árbol para determinar qué líneas de código han cambiado entre dos versiones de un archivo.
    - Se comparan las versiones previas y actuales de los archivos, verificando qué líneas han sido añadidas o eliminadas.
- Formatos de las Líneas:
    - Las líneas largas (más de 80 caracteres) son divididas y formateadas adecuadamente para no sobrepasar el límite de longitud.

Dependencias del Proyecto:

1. argparse: Manejo de argumentos de línea de comandos.
2. colorama: Para la coloración de texto en la consola.
3. pathlib: Para manejar rutas de archivos.
4. analizador_cambios.core.arbol.comparador_principal.ComparadorVersiones: Para comparar las versiones de archivos.
5. analizador_cambios.core.contadores.analizador.AnalizadorCodigo: Para analizar el código.
6. analizador_cambios.core.gestion_archivos.escribir_cambios.EscribirCambios: Para escribir los cambios en los archivos.
7. analizador_cambios.utils.formateador_linea.ExcepcionFormateo: Para el manejo de errores en el formateo.
8. contador_lineas.utils.archivo_utils.escribir_python: Para escribir archivos Python.
9. lineas_por_clase.core.gestion_archivos.almacenamiento_metricas.AlmacenamientoMetricas: Para almacenar métricas.
10. lineas_por_clase.utils.formateador_metricas.mostrar_tabla_metricas: Para mostrar las métricas en formato tabular.

Flujo de Trabajo

1. Inicialización: Se inicializa el entorno de colorama para la coloración de texto en la consola.
2. Procesamiento de Argumentos: Se procesan los argumentos de línea de comandos utilizando la función procesar_argumentos.
3. Almacenamiento de Métricas: Se crea una instancia de AlmacenamientoMetricas con la ruta de la base de datos proporcionada.
4. Caso Especial: Si se solicita mostrar todas las métricas (-tc) y no se proporcionan archivos, se muestran todas las métricas almacenadas y se termina la ejecución.
5. Validación de Argumentos: Se validan los argumentos utilizando la función validar_argumentos. Si hay errores, se muestran y se termina la ejecución.

6. Procesamiento de Archivos: Se procesan los archivos utilizando la función procesar_archivos. Si se solicita, se muestra la tabla de métricas.
7. Manejo de Excepciones: Se manejan las excepciones ExcepcionAnalizador, ExcepcionFormateo y Exception, mostrando mensajes de error en la consola.

Excepciones

- Excepción Analizador: Excepción personalizada para errores durante el análisis de código. Esta excepción se captura cuando ocurre un error durante el análisis de los archivos. Se imprime un mensaje de error en rojo.
- Excepción Formateo: Excepción personalizada para errores de formateo. Esta excepción se captura cuando ocurre un error durante el formateo de las líneas de código. Se imprime un mensaje de error en rojo.
- Exception: Excepción general para manejar cualquier otro tipo de error inesperado. Esta excepción general se captura para manejar cualquier otro tipo de error inesperado que pueda ocurrir durante la ejecución del programa. Se imprime un mensaje de error en rojo indicando que se trata de un error inesperado.

Ejemplo de uso:

Para comparar un archivo con otro

```
python .\src\analizador_cambios\__main__.py achivo1Analizar.py archivo2Analizar.py
```

Para visualizar los resultados

```
python .\src\analizador_cambios\__main__.py -tc
```

Diagrama de Secuencia