## Overview

This file provides the RepoAnalyzer class, a tool designed to analyze a local Git repository. Its primary function is to detect changes in code at a granular level, specifically identifying added, removed, and modified functions and classes within tracked files. It can analyze either staged changes or all uncommitted changes (staged and unstaged). The analysis respects ignore patterns and file extension filters defined in the application’s configuration.

## Classes

### RepoAnalyzer

Analyzes a Git repository for code changes at the object level (functions and classes).

The analyzer interfaces with a Git repository to compare the state of files between the HEAD commit and the current working state (either the index for staged changes or the working directory for all changes).

#### \_\_init\_\_(repo\_path: str)

Initializes the RepoAnalyzer instance.

It sets up the connection to the Git repository at the specified path. If the path is not a valid Git repository, it prints an error and raises a git.InvalidGitRepositoryError.

* **Parameters**:
  + repo\_path (str): The file system path to the Git repository.

#### \_is\_ignored(file\_path: str) -> bool

Checks if a given file path should be ignored based on the configured patterns.

This is a helper method that iterates through the CONFIG.ignore\_patterns list and uses fnmatch to check if the file path matches any of the patterns.

* **Parameters**:
  + file\_path (str): The path of the file to check.
* **Returns**:
  + bool: True if the file path matches an ignore pattern, False otherwise.

#### calculate\_fibonacci(n)

Calculates the n-th Fibonacci number using an iterative approach.

This function is designed to be a clear example of a new, distinct piece of logic added to test the documentation agent’s ability to intelligently place new documentation snippets.

* **Parameters**:
  + n (int): The position in the Fibonacci sequence.
* **Returns**:
  + int: The n-th Fibonacci number.

#### \_get\_file\_content(commit\_hash: str, file\_path: str) -> str

Retrieves the content of a file from a specific commit.

This helper method accesses the Git repository’s object database to read the content of a file as it existed in a particular commit.

* **Parameters**:
  + commit\_hash (str): The hash of the commit to retrieve the file from. Can also be a reference like “HEAD”.
  + file\_path (str): The path to the file within the repository.
* **Returns**:
  + str: The decoded content of the file. Returns an empty string if the file did not exist in that commit or if an error occurred.

#### analyze\_changes(staged\_only: bool = True) -> Dict[str, Dict[str, Any]]

Analyzes file changes to identify added, removed, and modified code objects (functions and classes).

This is the main method of the class. It diffs the repository to find changed files. For each changed file, it parses the old and new versions to extract function and class definitions, then compares these sets to determine what was added, removed, or modified.

* **Parameters**:
  + staged\_only (bool, optional): If True (default), analyzes only staged changes (HEAD vs. index). If False, analyzes all uncommitted changes, including staged, unstaged, and untracked files.
* **Returns**:
  + Dict[str, Dict[str, Any]]: A dictionary where keys are the file paths of changed files. Each value is another dictionary containing:
    - "status" (str): The change type (‘A’ for added, ‘M’ for modified, ‘D’ for deleted, etc.).
    - "added" (List[Dict]): A list of code objects that were added.
    - "removed" (List[Dict]): A list of code objects that were removed.
    - "modified" (List[Dict]): A list of code objects that were modified.

##### Usage Example

from code\_monitor.analyzer import RepoAnalyzer  
  
try:  
 # Initialize the analyzer with the path to your repo  
 analyzer = RepoAnalyzer('/path/to/your/git/repo')  
  
 # Analyze only the staged changes  
 staged\_changes = analyzer.analyze\_changes(staged\_only=True)  
  
 print("--- Staged Changes ---")  
 for file, changes in staged\_changes.items():  
 print(f"File: {file} (Status: {changes['status']})")  
 if changes['added']:  
 print(" Added:")  
 for item in changes['added']:  
 print(f" - {item['type']} {item['name']}")  
 if changes['modified']:  
 print(" Modified:")  
 for item in changes['modified']:  
 print(f" - {item['type']} {item['name']}")  
 if changes['removed']:  
 print(" Removed:")  
 for item in changes['removed']:  
 print(f" - {item['type']} {item['name']}")  
  
except git.InvalidGitRepositoryError:  
 print("Failed to initialize analyzer.")

This function analyzes a Git repository to identify code changes at the “object” level, such as functions, classes, or methods. It operates by comparing the Abstract Syntax Trees (ASTs) of files between different commits. This provides a more granular view of changes than a standard file-level diff, making it ideal for targeted code reviews, impact analysis, or automated changelog generation.

The analysis process involves: 1. Initializing a Git repository object from the provided path. 2. Iterating through the commits specified in the commit\_range. 3. For each commit, identifying modified Python files. 4. For each modified file, fetching the content of the file before and after the change. 5. Parsing both versions into code object trees using a language-specific parser. 6. Comparing the trees to pinpoint added, removed, or modified objects. 7. Aggregating these granular changes into a structured list for the entire commit range.

### Parameters

| Name | Type | Description |
| --- | --- | --- |
| repo\_path | str | The absolute or relative file system path to the local Git repository to be analyzed. |
| commit\_range | str | A string defining the range of commits to inspect, in a format accepted by git rev-list (e.g., 'HEAD~1..HEAD', 'main..develop'). |

### Returns

| Type | Description |
| --- | --- |
| List[Dict[str, any]] | A list of dictionaries, where each dictionary represents a single detected change to a code object. An empty list is returned if no changes are found. Each dictionary contains the following keys:**file\_path** (str): The relative path to the modified file within the repository.**object\_name** (str): The fully qualified name of the changed object (e.g., my\_module.MyClass.my\_method).**change\_type** (str): The type of change. Can be 'added', 'modified', or 'deleted'.**commit\_hash** (str): The full SHA of the commit where the change occurred. |

### Raises

| Type | Description |
| --- | --- |
| git.exc.InvalidGitRepositoryError | Raised if the provided repo\_path does not point to a valid Git repository. |
| ValueError | Raised if the commit\_range is invalid or cannot be resolved by Git. |

json { “added”: [ { “file\_path”: “src/utils.py”, “object\_name”: “new\_helper\_function”, “object\_type”: “function” } ], “removed”: [ { “file\_path”: “src/core.py”, “object\_name”: “OldApiClient”, “object\_type”: “class” } ], “modified”: [ { “file\_path”: “src/core.py”, “object\_name”: “process\_data”, “object\_type”: “function” } ] } ```

### Usage Example

```python from code\_monitor.analyzer import RepoAnalyzer

# Initialize the analyzer with the path to your repository

analyzer = RepoAnalyzer(repo\_path=‘/path/to/your/project’)

# Specify the commit range you want to analyze

old\_commit = ‘a1b2c3d4e5f67890’ new\_commit = ‘f0e9d8c7b6a54321’

# Perform the analysis to get a structured report of changes

changes = analyzer.analyze\_commits(old\_commit, new\_commit)

# Print the results

print(“— Code Object Changes —”) print(f”Analysis between {old\_commit[:7]} and {new\_commit[:7]}“)

print(f”Added objects: {len(changes[‘added’])}“) for item in changes[‘added’]: print(f” - [{item[‘object\_type’]}] {item[‘object\_name’]} in {item[‘file\_path’]}“)

print(f”objects: {len(changes[‘removed’])}“) for item in changes[‘removed’]: print(f” - [{item[‘object\_type’]}] {item[‘object\_name’]} in {item[‘file\_path’]}“)

print(f”objects: {len(changes[‘modified’])}“) for item in changes[‘modified’]: print(f” - [{item[‘object\_type’]}] {item[‘object\_name’]} in {item[‘file\_path’]}“)