## Overview  
  
This file provides the `RepoAnalyzer` class, a tool for analyzing a local Git repository to identify changes at the code object level (functions and classes). It can inspect staged changes or all uncommitted changes (staged, unstaged, and untracked files). The analyzer compares the state of files against the `HEAD` commit to determine which functions or classes have been added, removed, or modified, providing a more granular view of changes than a standard file-level diff.  
  
## Classes  
  
### RepoAnalyzer  
  
Analyzes a Git repository for code changes at the object level (functions and classes).  
  
The class initializes with a path to a Git repository. It provides methods to compare the current working or staged state against the last commit (`HEAD`) and identify specific code constructs that have changed.  
  
#### `\_\_init\_\_(repo\_path: str)`  
  
Initializes the `RepoAnalyzer` instance.  
  
- \*\*Parameters:\*\*  
 - `repo\_path` (str): The file system path to the Git repository to be analyzed.  
- \*\*Raises:\*\*  
 - `git.InvalidGitRepositoryError`: If the provided `repo\_path` is not a valid Git repository.  
  
---  
  
#### `\_is\_ignored(file\_path: str) -> bool`  
  
Checks if a file path matches any of the ignore patterns defined in the configuration (`CONFIG.ignore\_patterns`). This is a private helper method.  
  
- \*\*Parameters:\*\*  
 - `file\_path` (str): The path of the file to check.  
- \*\*Returns:\*\*  
 - `bool`: `True` if the file path matches an ignore pattern, `False` otherwise.  
  
---  
  
#### `\_get\_file\_content(commit\_hash: str, file\_path: str) -> str`  
  
Retrieves the content of a file from a specific commit in the repository. This is a private helper method.  
  
- \*\*Parameters:\*\*  
 - `commit\_hash` (str): The hash of the commit from which to retrieve the file content.  
 - `file\_path` (str): The path of the file within the repository.  
- \*\*Returns:\*\*  
 - `str`: The UTF-8 decoded content of the file. Returns an empty string if the file does not exist in the specified commit.  
  
---  
  
#### `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 determines the diff set based on the `staged\_only` flag. For each changed file, it parses the old and new versions to extract function and class definitions, then compares them to categorize the changes.  
  
- \*\*Parameters:\*\*  
 - `staged\_only` (bool, optional): If `True` (default), analyzes only staged changes (index vs. `HEAD`). If `False`, analyzes all uncommitted changes, including staged, unstaged (working tree vs. index), and untracked files.  
- \*\*Returns:\*\*  
 - `Dict[str, Dict[str, Any]]`: A dictionary where each key is a file path that has changed. The corresponding value is another dictionary containing:  
 - `"status"` (str): The change type ('A' for added, 'D' for deleted, 'M' for modified, etc.).  
 - `"added"` (List[str]): A list of names of functions/classes added to the file.  
 - `"removed"` (List[str]): A list of names of functions/classes removed from the file.  
 - `"modified"` (List[str]): A list of names of functions/classes that have been modified.  
  
##### Code Usage Example  
  
```python  
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(f" Added: {', '.join(changes['added'])}")  
 if changes['modified']:  
 print(f" Modified: {', '.join(changes['modified'])}")  
 if changes['removed']:  
 print(f" Removed: {', '.join(changes['removed'])}")  
  
 # Analyze all uncommitted changes  
 all\_changes = analyzer.analyze\_changes(staged\_only=False)  
 print("\n--- All Uncommitted Changes ---")  
 for file, changes in all\_changes.items():  
 print(f"File: {file} (Status: {changes['status']})")  
 if changes['added']:  
 print(f" Added: {', '.join(changes['added'])}")  
  
except git.InvalidGitRepositoryError:  
 print("The specified path is not a valid Git repository.")  
except Exception as e:  
 print(f"An error occurred: {e}")

<!-- DOC\_START: code\_monitor/analyzer.py::RepoAnalyzer -->  
### Class `RepoAnalyzer`  
  
Analyzes a Git repository to identify fine-grained code changes at the Python object level (e.g., functions, classes, methods) for a specific commit.  
  
This class leverages the `gitpython` library to inspect commit diffs and Python's built-in `ast` (Abstract Syntax Tree) module to parse source code. It identifies which specific functions or classes within a file were modified, added, or had their type changed.  
  
#### `\_\_init\_\_(self, repo\_path: str, context\_lines: int = 3)`  
  
Initializes the `RepoAnalyzer` with the path to a local Git repository.  
  
\*\*Parameters:\*\*  
  
| Name | Type | Description | Default |  
| --------------- | ---- | -------------------------------------------------------------------------------------------------------- | ------- |  
| `repo\_path` | `str` | The file system path to the Git repository to be analyzed. | |  
| `context\_lines` | `int` | The number of lines of context to include around a detected change within a code object. | `3` |  
  
---  
  
#### `analyze\_commit(self, commit\_hash: str) -> list[CodeChange]`  
  
Analyzes a single commit and returns a list of all identified code object changes.  
  
The method works by comparing the specified commit against its first parent. For the initial commit in a repository (which has no parents), it treats all files as newly added. It processes files that were added, modified, or had their type changed, ignoring deleted files.  
  
For each relevant file, it parses the code from the \*parent\* commit to build an Abstract Syntax Tree (AST). This tree is used to identify the boundaries of every class and function. The analyzer then checks the diff from the commit to determine which of these specific code objects have been altered.  
  
The analysis gracefully skips files that are not valid Python code (e.g., raise a `SyntaxError` during parsing) or have decoding issues.  
  
\*\*Parameters:\*\*  
  
| Name | Type | Description |  
| ------------- | ---- | -------------------------------------------------- |  
| `commit\_hash` | `str` | The full or short SHA hash of the commit to analyze. |  
  
\*\*Returns:\*\*  
  
| Type | Description |  
| -------------------- | ----------------------------------------------------------------------------------------------------------------------------------------------------------------------- |  
| `list[CodeChange]` | A list of `CodeChange` objects, where each object represents a distinct modification to a single code object (function, class, etc.) within a file. Returns an empty list if no changes are found. |  
  
\*\*Example Usage:\*\*  
  
```python  
# Assuming CodeChange is a dataclass that holds change details  
from dataclasses import dataclass  
  
@dataclass  
class CodeChange:  
 file\_path: str  
 object\_name: str  
 # ... other attributes  
  
# Initialize the analyzer  
analyzer = RepoAnalyzer(repo\_path="/path/to/your/project")  
  
# Analyze a specific commit  
commit\_sha = "a1b2c3d4e5f6a1b2c3d4e5f6"  
changes = analyzer.analyze\_commit(commit\_sha)  
  
# Print the results  
if changes:  
 print(f"Found {len(changes)} object-level changes in commit {commit\_sha[:7]}:")  
 for change in changes:  
 print(f"- {change.file\_path}: Modified object '{change.object\_name}'")  
else:  
 print(f"No analyzable Python code changes found in commit {commit\_sha[:7]}.")