**Python Code Analysis Report**

# Summary

## Files Analyzed

• app.py: 1 functions

• main.py: 2 functions

• tests/test\_analyzer.py: 4 functions

• src/analyzer.py: 1 functions

• src/action.py: 1 functions

• src/code\_explainer/llm\_integration.py: 1 functions

• src/code\_explainer/cli.py: 1 functions

• src/code\_explainer/document\_generator.py: 1 functions

• src/code\_explainer/code\_analyzer.py: 1 functions

Total elements found: 13

* • Functions: 13

# Code Structure

## File: app.py

### Function: main (Lines 7-77)

Documentation:

Main application function.

Source Code:

def main():  
 """Main application function."""  
 st.set\_page\_config(  
 page\_title="Python Code Explainer",  
 page\_icon="🤖",  
 layout="wide"  
 )  
   
 st.title("Python Code Explainer")  
 st.caption("Upload a Python file to analyze its structure and get explanations")  
   
 # Sidebar for settings  
 with st.sidebar:  
 st.header("Settings")  
 model\_name = st.selectbox(  
 "LLM Model",  
 ["llama2", "llama3", "mistral"],  
 index=0,  
 help="Select the language model to use for generating explanations"  
 )  
   
 # File uploader  
 uploaded\_file = st.file\_uploader("Upload Python (.py) file", type="py")  
   
 if uploaded\_file:  
 try:  
 # Read and analyze the code  
 code = uploaded\_file.read().decode()  
 code\_elements = extract\_elements(code)  
   
 # Display code analysis  
 st.header("Code Analysis")  
 for el in code\_elements:  
 with st.expander(f"{el['type']}: {el['name']} (Lines {el['start\_line']}-{el['end\_line']})"):  
 st.caption(f"Location: Lines {el['start\_line']}-{el['end\_line']}")  
   
 if el['args']:  
 st.write(f"\*\*Arguments:\*\* `{', '.join(el['args'])}`")  
 if el['type'] != 'Class' and el['has\_return']:  
 st.write("\*\*Returns:\*\* Yes")  
   
 if el['docstring']:  
 st.subheader("Documentation")  
 st.text(el['docstring'])  
   
 st.subheader("Source Code")  
 st.code(el['source'], language='python')  
   
 # Generate explanation  
 if st.button("Generate Explanation", type="primary"):  
 with st.spinner("Analyzing code with AI..."):  
 try:  
 explanation = generate\_explanation(code\_elements, model=model\_name)  
 st.success("AI Explanation:")  
 st.write(explanation)  
   
 # Generate and offer download of DOCX report  
 doc\_buffer = create\_document(code\_elements, explanation)  
 if doc\_buffer:  
 st.download\_button(  
 label="📥 Download Analysis Report",  
 data=doc\_buffer,  
 file\_name="code\_analysis\_report.docx",  
 mime="application/vnd.openxmlformats-officedocument.wordprocessingml.document"  
 )  
   
 except Exception as e:  
 st.error(f"Error generating explanation: {str(e)}")  
   
 except Exception as e:  
 st.error(f"Error processing file: {str(e)}")

----------------------------------------

## File: main.py

### Function: extract\_elements (Lines 14-47)

Arguments: code

Returns: Yes

Documentation:

Extracts all top-level classes and functions with docstrings and source code.

Source Code:

def extract\_elements(code):  
 """Extracts all top-level classes and functions with docstrings and source code."""  
 tree = ast.parse(code)  
 elements = []  
 for node in tree.body:  
 if isinstance(node, (ast.FunctionDef, ast.ClassDef, ast.AsyncFunctionDef)):  
 # Get basic info  
 element\_type = node.\_\_class\_\_.\_\_name\_\_  
 name = node.name  
 doc = ast.get\_docstring(node) or ""  
   
 # Get full source code  
 src = ast.get\_source\_segment(code, node) or ""  
   
 # Get line numbers  
 start\_line = getattr(node, 'lineno', 0)  
 end\_line = getattr(node, 'end\_lineno', start\_line)  
   
 # Get function arguments if it's a function  
 args = []  
 if hasattr(node, 'args'):  
 args = [arg.arg for arg in node.args.args]  
   
 elements.append({  
 'type': element\_type.replace('Def', ''), # 'FunctionDef' -> 'Function', 'ClassDef' -> 'Class'  
 'name': name,  
 'docstring': doc,  
 'source': src,  
 'start\_line': start\_line,  
 'end\_line': end\_line,  
 'args': args if args else None,  
 'has\_return': any(isinstance(n, ast.Return) for n in ast.walk(node))  
 })  
 return elements

----------------------------------------

### Function: llama\_explain (Lines 49-148)

Arguments: code\_elements

Returns: Yes

Documentation:

Send code info to Llama via Ollama and get detailed non-technical summary.  
  
The prompt is structured as follows:  
  
[STATIC] - These parts never change:  
- The initial instructions to the AI about its role and how to respond  
- The section headers (e.g., 'Documentation:', 'Code:')  
- The formatting of the response  
  
[DYNAMIC] - These parts are filled with actual code analysis:  
- Function/Class names and types  
- Line numbers where code appears  
- Documentation strings from the code  
- Function arguments (if any)  
- Return value indicators  
- The actual source code  
  
Example of how the prompt will look:  
  
[SYSTEM PROMPT - Static]  
You are a helpful assistant. Read the information below and summarize what this Python code does,  
explaining it in detail, in plain, non-technical English, to a non-programmer. Avoid jargon.  
Where possible, use analogies and concrete examples.  
  
Here is the code to summarize:  
  
[DYNAMIC CONTENT - Example for one function]  
Function 'calculate\_total':  
Location: Lines 5-10  
Documentation: Calculates the total price including tax  
Arguments: price, tax\_rate  
Returns: Yes  
Code:  
def calculate\_total(price, tax\_rate):

Source Code:

def llama\_explain(code\_elements):  
 """  
 Send code info to Llama via Ollama and get detailed non-technical summary.  
   
 The prompt is structured as follows:  
   
 [STATIC] - These parts never change:  
 - The initial instructions to the AI about its role and how to respond  
 - The section headers (e.g., 'Documentation:', 'Code:')  
 - The formatting of the response  
   
 [DYNAMIC] - These parts are filled with actual code analysis:  
 - Function/Class names and types  
 - Line numbers where code appears  
 - Documentation strings from the code  
 - Function arguments (if any)  
 - Return value indicators  
 - The actual source code  
   
 Example of how the prompt will look:  
   
 [SYSTEM PROMPT - Static]  
 You are a helpful assistant. Read the information below and summarize what this Python code does,  
 explaining it in detail, in plain, non-technical English, to a non-programmer. Avoid jargon.  
 Where possible, use analogies and concrete examples.  
   
 Here is the code to summarize:  
   
 [DYNAMIC CONTENT - Example for one function]  
 Function 'calculate\_total':  
 Location: Lines 5-10  
 Documentation: Calculates the total price including tax  
 Arguments: price, tax\_rate  
 Returns: Yes  
 Code:  
 def calculate\_total(price, tax\_rate):  
 """  
 # Compose prompt with enhanced information  
 combined = "\n\n".join(  
 f"{el['type']} '{el['name']}':\n"  
 f"Location: Lines {el['start\_line']}-{el['end\_line']}\n"  
 f"Documentation: {el['docstring']}\n"  
 + (f"Arguments: {', '.join(el['args'])}\n" if el['args'] else "")   
 + ("Returns: Yes\n" if el['has\_return'] and el['type'] != 'Class' else "")  
 + f"Code:\n{el['source'] or ''}"  
 for el in code\_elements  
 )  
   
 system\_prompt = (  
 "You are a helpful assistant. Your job is to explain the Python code and workflow described below "  
 "in plain, non-technical English to someone without a programming background. Avoid technical jargon. "  
 "Use relatable analogies and simple examples where appropriate.\n\n"  
  
 "📌 Task Overview:\n"  
 "- Break down the Python code into understandable parts.\n"  
 "- Present the explanation in a two-column table:\n"  
 " 1. Section of the prompt\n"  
 " 2. Whether it is dynamic or static\n\n"  
  
 "🧠 User Context:\n"  
 "- The user is building an automated assessment tool using LLMs.\n"  
 "- The tool generates subtopics and test questions from a topic + grade + learning objective.\n"  
 "- They want help modifying Prompt 1 to include a new variable: the learning objective.\n\n"  
  
 "💡 Input Example:\n"  
 " topic = 'Ratios and Proportional Relationships'\n"  
 " student\_class = '6th standard'\n"  
 " learning\_objective = 'Understand ratio concepts and use ratio reasoning to solve problems.'\n\n"  
  
 "📝 Full Prompt 1 (Subtopic Generator):\n"  
 "I want a list of sub-topics for the topic \"{topic}\" which is taught to a \"{student\_class}\" student "  
 "with a learning objective \"{learning\_objective}\".\n"  
 "First, output the learning objective exactly as given.\n"  
 "Then, output the sub-topics ONLY as a Python list. Do not include any commentary or explanation.\n\n"  
  
 "🔧 System Context:\n"  
 "We are building an automated assessment web app where questions are usually uploaded manually into a MySQL database. "  
 "This tool uses large language models to generate those questions automatically, saving time and effort.\n\n"  
  
 "📋 Additional User Requests:\n"  
 "- Modify the original code to include the new learning objective variable.\n"  
 "- Ensure that generate\_questions\_for\_subtopic also uses the learning objective.\n"  
 "- Recreate the dynamic/static breakdown table of Prompt 1 and Prompt 2.\n"  
 "- Show an example of what Prompt 1 and Prompt 2 look like after the code runs.\n"  
 "- Give a step-by-step guide to feeding these prompts into a ChatGPT conversation.\n"  
 "- Convert the instructions into clean documentation.\n"  
 "- Provide a downloadable .docx version of the documentation.\n\n"  
  
 f"{combined}\n\n"  
 "🧾 Now, please provide a detailed, beginner-friendly explanation:"  
)  
  
  
 try:  
 client = Client(host='http://localhost:11434')  
 response = client.chat(model='llama2', messages=[{"role": "user", "content": system\_prompt}])  
   
 except Exception as e:  
 return f"Error generating explanation: {str(e)}"  
 return response['message']['content']

----------------------------------------

## File: tests/test\_analyzer.py

### Function: test\_extract\_elements\_with\_functions (Lines 5-18)

Documentation:

Test that extract\_elements can parse a simple function.

Source Code:

def test\_extract\_elements\_with\_functions():  
 """Test that extract\_elements can parse a simple function."""  
 code = """  
def hello(name: str) -> str:  
 \"\"\"Return a greeting message.\"\"\"  
 return f"Hello, {name}!"  
"""  
 elements = extract\_elements(code)  
 assert len(elements) == 1  
 assert elements[0]['name'] == "hello"  
 assert elements[0]['type'] == "Function"  
 assert "Return a greeting message" in elements[0]['docstring']  
 assert elements[0]['args'] == ['name']  
 assert elements[0]['has\_return'] is True

----------------------------------------

### Function: test\_extract\_elements\_with\_class (Lines 20-35)

Documentation:

Test that extract\_elements can parse a class definition.

Source Code:

def test\_extract\_elements\_with\_class():  
 """Test that extract\_elements can parse a class definition."""  
 code = """  
class Greeter:  
 \"\"\"A class that greets people.\"\"\"  
   
 def \_\_init\_\_(self, name: str):  
 self.name = name  
   
 def greet(self) -> str:  
 \"\"\"Return a greeting.\"\"\"  
 return f"Hello, {self.name}!"  
"""  
 elements = extract\_elements(code)  
 assert len(elements) == 1 # Only the class itself is extracted  
 assert elements[0]['type'] == 'Class' and elements[0]['name'] == 'Greeter'

----------------------------------------

### Function: sample\_code (Lines 38-48)

Returns: Yes

Source Code:

def sample\_code():  
 return """  
def add(a: int, b: int) -> int:  
 \"\"\"Add two numbers.\"\"\"  
 return a + b  
   
class Calculator:  
 def multiply(self, x: float, y: float) -> float:  
 \"\"\"Multiply two numbers.\"\"\"  
 return x \* y  
"""

----------------------------------------

### Function: test\_extract\_elements\_with\_fixture (Lines 50-55)

Arguments: sample\_code

Documentation:

Test extract\_elements using a fixture.

Source Code:

def test\_extract\_elements\_with\_fixture(sample\_code):  
 """Test extract\_elements using a fixture."""  
 elements = extract\_elements(sample\_code)  
 assert len(elements) == 2 # add function and Calculator class  
 assert any(e['name'] == 'add' and e['type'] == 'Function' for e in elements)  
 assert any(e['name'] == 'Calculator' and e['type'] == 'Class' for e in elements)

----------------------------------------

## File: src/analyzer.py

### Function: extract\_elements (Lines 4-31)

Arguments: code

Returns: Yes

Documentation:

Extracts all top-level classes and functions with docstrings and source code.

Source Code:

def extract\_elements(code: str) -> List[Dict[str, Any]]:  
 """Extracts all top-level classes and functions with docstrings and source code."""  
 tree = ast.parse(code)  
 elements = []  
 for node in tree.body:  
 if isinstance(node, (ast.FunctionDef, ast.ClassDef, ast.AsyncFunctionDef)):  
 element\_type = node.\_\_class\_\_.\_\_name\_\_  
 name = node.name  
 doc = ast.get\_docstring(node) or ""  
 src = ast.get\_source\_segment(code, node) or ""  
 start\_line = getattr(node, 'lineno', 0)  
 end\_line = getattr(node, 'end\_lineno', start\_line)  
   
 args = []  
 if hasattr(node, 'args'):  
 args = [arg.arg for arg in node.args.args]  
   
 elements.append({  
 'type': element\_type.replace('Def', ''),  
 'name': name,  
 'docstring': doc,  
 'source': src,  
 'start\_line': start\_line,  
 'end\_line': end\_line,  
 'args': args if args else None,  
 'has\_return': any(isinstance(n, ast.Return) for n in ast.walk(node))  
 })  
 return elements

----------------------------------------

## File: src/action.py

### Function: analyze\_repository (Lines 11-86)

Returns: Yes

Documentation:

Main function to analyze the repository.

Source Code:

def analyze\_repository() -> None:  
 """Main function to analyze the repository."""  
 # Get environment variables  
 ollama\_model = os.getenv('OLLAMA\_MODEL', 'llama2')  
 ollama\_host = os.getenv('OLLAMA\_HOST', 'http://localhost:11434')  
   
 # Find all Python files  
 python\_files = list(Path('.').rglob('\*.py'))  
   
 all\_elements = []  
 for py\_file in python\_files:  
 try:  
 with open(py\_file, 'r', encoding='utf-8') as f:  
 code = f.read()  
 elements = extract\_elements(code)  
 for el in elements:  
 el['file'] = str(py\_file)  
 all\_elements.extend(elements)  
 except Exception as e:  
 print(f"Error processing {py\_file}: {e}", file=sys.stderr)  
   
 if not all\_elements:  
 print("No code elements found to analyze.")  
 return  
   
 # Create docs directory if it doesn't exist  
 docs\_dir = Path('docs')  
 docs\_dir.mkdir(exist\_ok=True)  
   
 # Format elements for the prompt  
 combined = "\n\n".join(  
 f"File: {el['file']}\n"  
 f"{el['type']} '{el['name']}':\n"  
 f"Location: Lines {el['start\_line']}-{el['end\_line']}\n"  
 f"Documentation: {el['docstring']}\n"  
 + (f"Arguments: {', '.join(el['args'])}\n" if el['args'] else "")   
 + ("Returns: Yes\n" if el['has\_return'] and el['type'] != 'Class' else "")  
 + f"Code:\n{el['source'] or ''}"  
 for el in all\_elements  
 )  
   
 # Generate explanation  
 client = Client(host=ollama\_host)  
 system\_prompt = """You are a helpful assistant. Your job is to explain the Python code and workflow described below   
in plain, non-technical English to someone without a programming background. Avoid technical jargon.   
Use relatable analogies and simple examples where appropriate.  
  
{combined}  
"""  
 try:  
 response = client.chat(  
 model=ollama\_model,  
 messages=[{"role": "user", "content": system\_prompt}],  
 stream=False  
 )  
 explanation = response['message']['content']  
 print("\n" + "="\*80)  
 print("CODE ANALYSIS REPORT")  
 print("="\*80)  
 print(explanation)  
   
 # Generate and save the document  
 timestamp = datetime.now().strftime("%Y%m%d\_%H%M%S")  
 doc\_path = docs\_dir / f"code\_analysis\_{timestamp}.docx"  
   
 doc\_buffer = create\_document(all\_elements, explanation, "Python Code Analysis Report")  
 if doc\_buffer:  
 with open(doc\_path, 'wb') as f:  
 f.write(doc\_buffer.getvalue())  
 print(f"\nDocument generated: {doc\_path}")  
 else:  
 print("\nWarning: Could not generate Word document. Make sure python-docx is installed.")  
   
 except Exception as e:  
 print(f"Error during analysis: {e}", file=sys.stderr)  
 sys.exit(1)

----------------------------------------

## File: src/code\_explainer/llm\_integration.py

### Function: generate\_explanation (Lines 7-112)

Arguments: code\_elements, model

Returns: Yes

Documentation:

Generate a non-technical explanation of the code using the specified LLM.  
  
Args:  
 code\_elements: List of code element dictionaries from extract\_elements()  
 Each element may contain a 'file' key indicating its source file.  
 model: Name of the LLM model to use (default: "llama2")  
   
Returns:  
 Generated explanation as a string

Source Code:

def generate\_explanation(code\_elements: List[Dict[str, Any]], model: str = "llama2") -> str:  
 """  
 Generate a non-technical explanation of the code using the specified LLM.  
   
 Args:  
 code\_elements: List of code element dictionaries from extract\_elements()  
 Each element may contain a 'file' key indicating its source file.  
 model: Name of the LLM model to use (default: "llama2")  
   
 Returns:  
 Generated explanation as a string  
 """  
 if not code\_elements:  
 return "No code elements to analyze."  
   
 try:  
 # Group elements by file  
 elements\_by\_file = {}  
 for el in code\_elements:  
 file\_name = el.get('file', 'main.py')  
 if file\_name not in elements\_by\_file:  
 elements\_by\_file[file\_name] = []  
 elements\_by\_file[file\_name].append(el)  
   
 # Compile code elements into a formatted string, grouped by file  
 combined\_parts = []  
 for file\_name, elements in elements\_by\_file.items():  
 file\_section = f"# File: {file\_name}\n\n"  
   
 for el in elements:  
 element\_section = (  
 f"## {el['type']} '{el['name']}'\n"  
 f"Location: Lines {el['start\_line']}-{el['end\_line']}\n"  
 )  
   
 if el['docstring']:  
 element\_section += f"Documentation: {el['docstring']}\n"  
   
 if el['args']:  
 element\_section += f"Arguments: {', '.join(el['args'])}\n"  
   
 if el['type'] != 'Class' and el['has\_return']:  
 element\_section += "Returns: Yes\n"  
   
 element\_section += f"Code:\n```python\n{el['source'] or ''}\n```\n\n"  
 file\_section += element\_section  
   
 combined\_parts.append(file\_section)  
   
 combined = "\n".join(combined\_parts)  
   
 system\_prompt = (  
 "You are a helpful assistant. Your job is to explain the Python code and workflow described below "  
 "in plain, non-technical English to someone without a programming background. Avoid technical jargon. "  
 "Use relatable analogies and simple examples where appropriate.\n\n"  
  
 "📌 Task Overview:\n"  
 "- Break down the Python code into understandable parts.\n"  
 "- Present the explanation in a two-column table:\n"  
 " 1. Section of the prompt\n"  
 " 2. Whether it is dynamic or static\n\n"  
  
 "🧠 User Context:\n"  
 "- The user is building an automated assessment tool using LLMs.\n"  
 "- The tool generates subtopics and test questions from a topic + grade + learning objective.\n"  
 "- They want help modifying Prompt 1 to include a new variable: the learning objective.\n\n"  
  
 "💡 Input Example:\n"  
 " topic = 'Ratios and Proportional Relationships'\n"  
 " student\_class = '6th standard'\n"  
 " learning\_objective = 'Understand ratio concepts and use ratio reasoning to solve problems.'\n\n"  
  
 "📝 Full Prompt 1 (Subtopic Generator):\n"  
 "I want a list of sub-topics for the topic \"{topic}\" which is taught to a \"{student\_class}\" student "  
 "with a learning objective \"{learning\_objective}\".\n"  
 "First, output the learning objective exactly as given.\n"  
 "Then, output the sub-topics ONLY as a Python list. Do not include any commentary or explanation.\n\n"  
  
 "🔧 System Context:\n"  
 "We are building an automated assessment web app where questions are usually uploaded manually into a MySQL database. "  
 "This tool uses large language models to generate those questions automatically, saving time and effort.\n\n"  
  
 "📋 Additional User Requests:\n"  
 "- Modify the original code to include the new learning objective variable.\n"  
 "- Ensure that generate\_questions\_for\_subtopic also uses the learning objective.\n"  
 "- Recreate the dynamic/static breakdown table of Prompt 1 and Prompt 2.\n"  
 "- Show an example of what Prompt 1 and Prompt 2 look like after the code runs.\n"  
 "- Give a step-by-step guide to feeding these prompts into a ChatGPT conversation.\n"  
 "- Convert the instructions into clean documentation.\n"  
 "- Provide a downloadable .docx version of the documentation.\n\n"  
  
 f"{combined}\n\n"  
 "🧾 Now, please provide a detailed, beginner-friendly explanation:"  
)  
   
 client = Client(host='http://localhost:11434')  
 response = client.chat(  
 model=model,  
 messages=[{"role": "user", "content": system\_prompt}],  
 stream=False  
 )  
   
 return response['message']['content']  
   
 except Exception as e:  
 raise RuntimeError(f"Error generating explanation: {str(e)}")

----------------------------------------

## File: src/code\_explainer/cli.py

### Function: main (Lines 9-120)

Arguments: args

Returns: Yes

Documentation:

Main entry point for the CLI.  
  
Args:  
 args: Command-line arguments (defaults to sys.argv[1:])  
   
Returns:  
 int: Exit code (0 for success, non-zero for error)

Source Code:

def main(args: Optional[list] = None) -> int:  
 """  
 Main entry point for the CLI.  
   
 Args:  
 args: Command-line arguments (defaults to sys.argv[1:])  
   
 Returns:  
 int: Exit code (0 for success, non-zero for error)  
 """  
 parser = argparse.ArgumentParser(description="Python Code Explainer")  
 parser.add\_argument(  
 "path",  
 type=str,  
 help="Python file or directory to analyze"  
 )  
 parser.add\_argument(  
 "--recursive",  
 "-r",  
 action="store\_true",  
 help="Recursively process Python files in subdirectories"  
 )  
 parser.add\_argument(  
 "--model",  
 "-m",  
 type=str,  
 default="llama2",  
 help="LLM model to use (default: llama2)"  
 )  
 parser.add\_argument(  
 "--output",  
 "-o",  
 type=str,  
 help="Output file for the report (default: print to console)"  
 )  
   
 parsed\_args = parser.parse\_args(args)  
   
 try:  
 # Import here to avoid loading everything when the CLI is not used  
 from .code\_analyzer import extract\_elements  
 from .llm\_integration import generate\_explanation  
 from .document\_generator import create\_document  
   
 # Process the input path (file or directory)  
 path = Path(parsed\_args.path)  
 if not path.exists():  
 print(f"Error: Path '{path}' not found", file=sys.stderr)  
 return 1  
  
 # Collect all Python files to process  
 python\_files = []  
 if path.is\_file() and path.suffix == '.py':  
 python\_files = [path]  
 elif path.is\_dir():  
 pattern = '\*\*/\*.py' if parsed\_args.recursive else '\*.py'  
 python\_files = list(path.glob(pattern))  
 if not python\_files:  
 print(f"No Python files found in {path}")  
 return 0  
 else:  
 print(f"Error: Path must be a Python file or directory")  
 return 1  
  
 all\_code\_elements = []  
 for py\_file in python\_files:  
 try:  
 print(f"Processing {py\_file}...", file=sys.stderr)  
 code = py\_file.read\_text(encoding="utf-8")  
 elements = extract\_elements(code)  
 for el in elements:  
 el['file'] = str(py\_file.relative\_to(path.parent))  
 el['file\_path'] = str(py\_file)  
 all\_code\_elements.extend(elements)  
 except Exception as e:  
 print(f"Error processing {py\_file}: {e}", file=sys.stderr)  
  
 if not all\_code\_elements:  
 print("No code elements found in any files.")  
 return 0  
   
 # Generate explanation  
 print("\nAnalyzing code...", file=sys.stderr)  
 explanation = generate\_explanation(all\_code\_elements, model=parsed\_args.model)  
   
 # Output the result  
 if parsed\_args.output:  
 output\_path = Path(parsed\_args.output)  
 if output\_path.suffix.lower() == '.docx':  
 buffer = create\_document(all\_code\_elements, explanation)  
 if buffer:  
 output\_path.write\_bytes(buffer.getvalue())  
 print(f"Report saved to {output\_path}")  
 else:  
 print("Error: Could not generate Word document. Is python-docx installed?", file=sys.stderr)  
 return 1  
 else:  
 with output\_path.open('w', encoding='utf-8') as f:  
 f.write("# Code Analysis Report\n\n")  
 f.write(explanation)  
 print(f"Report saved to {output\_path}")  
 else:  
 print("\n" + "="\*80)  
 print("CODE ANALYSIS REPORT")  
 print("="\*80)  
 print(explanation)  
   
 return 0  
   
 except Exception as e:  
 print(f"Error: {str(e)}", file=sys.stderr)  
 return 1

----------------------------------------

## File: src/code\_explainer/document\_generator.py

### Function: create\_document (Lines 15-153)

Arguments: code\_elements, explanation, title

Returns: Yes

Documentation:

Create a Word document from code analysis and explanation.  
  
Args:  
 code\_elements: List of code element dictionaries, each can include 'file' key  
 explanation: Generated explanation text  
 title: Title for the document  
   
Returns:  
 BytesIO buffer containing the document, or None if docx is not available

Source Code:

def create\_document(code\_elements: List[Dict[str, Any]], explanation: str,   
 title: str = "Python Code Analysis") -> Optional[BytesIO]:  
 """  
 Create a Word document from code analysis and explanation.  
   
 Args:  
 code\_elements: List of code element dictionaries, each can include 'file' key  
 explanation: Generated explanation text  
 title: Title for the document  
   
 Returns:  
 BytesIO buffer containing the document, or None if docx is not available  
 """  
 if not DOCX\_AVAILABLE:  
 return None  
   
 def \_get\_or\_create\_code\_style(doc):  
 """Get the Code style or create it if it doesn't exist."""  
 try:  
 return doc.styles['Code']  
 except KeyError:  
 code\_style = doc.styles.add\_style('Code', 1) # 1 = WD\_STYLE\_TYPE.PARAGRAPH  
 code\_style.font.name = 'Courier New'  
 code\_style.font.size = Pt(10)  
 code\_style.paragraph\_format.space\_after = Pt(6)  
 return code\_style  
   
 try:  
 doc = Document()  
   
 # Add title  
 title\_para = doc.add\_heading(level=0)  
 title\_run = title\_para.add\_run(title)  
 title\_run.bold = True  
 title\_para.alignment = WD\_PARAGRAPH\_ALIGNMENT.CENTER  
   
 # Ensure we have the Code style  
 \_get\_or\_create\_code\_style(doc)  
   
 # Add summary section  
 doc.add\_heading("Summary", level=1)  
   
 # Count elements by type and file  
 elements\_by\_file = {}  
 elements\_by\_type = {}  
   
 for el in code\_elements:  
 file\_name = el.get('file', 'unknown.py')  
 el\_type = el.get('type', 'Unknown')  
   
 # Count by file  
 if file\_name not in elements\_by\_file:  
 elements\_by\_file[file\_name] = {'functions': 0, 'classes': 0, 'async\_functions': 0}  
   
 if el\_type == 'Function':  
 elements\_by\_file[file\_name]['functions'] += 1  
 elif el\_type == 'AsyncFunction':  
 elements\_by\_file[file\_name]['async\_functions'] += 1  
 elif el\_type == 'Class':  
 elements\_by\_file[file\_name]['classes'] += 1  
   
 # Count by type  
 if el\_type not in elements\_by\_type:  
 elements\_by\_type[el\_type] = 0  
 elements\_by\_type[el\_type] += 1  
   
 # Add summary table  
 if elements\_by\_file:  
 doc.add\_heading("Files Analyzed", level=2)  
 for file, counts in elements\_by\_file.items():  
 parts = []  
 if counts['classes']:  
 parts.append(f"{counts['classes']} classes")  
 if counts['functions']:  
 parts.append(f"{counts['functions']} functions")  
 if counts['async\_functions']:  
 parts.append(f"{counts['async\_functions']} async functions")  
 doc.add\_paragraph(f"• {file}: {', '.join(parts) if parts else 'No elements found'}")  
   
 # Add element type summary  
 if elements\_by\_type:  
 doc.add\_paragraph(f"Total elements found: {sum(elements\_by\_type.values())}")  
 for el\_type, count in elements\_by\_type.items():  
 doc.add\_paragraph(f"• {el\_type}s: {count}", style='List Bullet')  
   
 # Add code structure section  
 doc.add\_heading("Code Structure", level=1)  
   
 # Group elements by file  
 elements\_by\_file = {}  
 for el in code\_elements:  
 file\_name = el.get('file', 'unknown.py')  
 if file\_name not in elements\_by\_file:  
 elements\_by\_file[file\_name] = []  
 elements\_by\_file[file\_name].append(el)  
   
 # Process each file  
 for file\_name, elements in elements\_by\_file.items():  
 # Add file header  
 doc.add\_heading(f"File: {file\_name}", level=2)  
   
 for el in elements:  
 # Add element header  
 el\_header = f"{el['type']}: {el['name']} (Lines {el['start\_line']}-{el['end\_line']})"  
 doc.add\_heading(el\_header, level=3)  
   
 # Add metadata  
 if el['args']:  
 doc.add\_paragraph(f"Arguments: {', '.join(el['args'])}")  
   
 if el['type'] != 'Class' and el['has\_return']:  
 doc.add\_paragraph("Returns: Yes")  
   
 if el['docstring']:  
 doc.add\_paragraph("Documentation:")  
 doc.add\_paragraph(el['docstring'], style='Intense Quote')  
   
 # Add source code  
 doc.add\_paragraph("Source Code:")  
 doc.add\_paragraph(el['source'], style='Code')  
   
 # Add a small separator between elements  
 doc.add\_paragraph()  
 doc.add\_paragraph("-" \* 40)  
 doc.add\_paragraph()  
   
 # Add explanation section  
 doc.add\_heading("AI Explanation", level=1)  
 doc.add\_paragraph(explanation)  
   
 # Save to buffer  
 buffer = BytesIO()  
 doc.save(buffer)  
 buffer.seek(0)  
   
 return buffer  
   
 except Exception as e:  
 raise RuntimeError(f"Error generating document: {str(e)}")

----------------------------------------

## File: src/code\_explainer/code\_analyzer.py

### Function: extract\_elements (Lines 8-50)

Arguments: code

Returns: Yes

Documentation:

Extract all top-level classes and functions with their metadata from Python code.  
  
Args:  
 code: Python source code as a string  
   
Returns:  
 List of dictionaries containing information about each code element

Source Code:

def extract\_elements(code: str) -> List[Dict[str, Any]]:  
 """  
 Extract all top-level classes and functions with their metadata from Python code.  
   
 Args:  
 code: Python source code as a string  
   
 Returns:  
 List of dictionaries containing information about each code element  
 """  
 try:  
 tree = ast.parse(code)  
 elements = []  
   
 for node in tree.body:  
 if isinstance(node, (ast.FunctionDef, ast.ClassDef, ast.AsyncFunctionDef)):  
 element\_type = node.\_\_class\_\_.\_\_name\_\_.replace('Def', '')  
 name = node.name  
 doc = ast.get\_docstring(node) or ""  
 src = ast.get\_source\_segment(code, node) or ""  
   
 start\_line = getattr(node, 'lineno', 0)  
 end\_line = getattr(node, 'end\_lineno', start\_line)  
   
 args = []  
 if hasattr(node, 'args') and hasattr(node.args, 'args'):  
 args = [arg.arg for arg in node.args.args]  
   
 elements.append({  
 'type': element\_type,  
 'name': name,  
 'docstring': doc,  
 'source': src,  
 'start\_line': start\_line,  
 'end\_line': end\_line,  
 'args': args if args else None,  
 'has\_return': any(isinstance(n, ast.Return) for n in ast.walk(node))  
 })  
   
 return elements  
   
 except SyntaxError as e:  
 raise ValueError(f"Error parsing Python code: {e}")

----------------------------------------

# AI Explanation

Of course! I'd be happy to help you understand the Python code and workflow described below. Let me break it down for you in plain English:  
  
The code is a series of instructions that tell a computer what to do with some data. In this case, the data is a list of numbers called "prices." The code is like a recipe for a computer program that takes the prices as input and produces some output.  
  
Here's what the code does:  
  
1. It starts by asking the computer to take the prices and put them in a special container called a "list." This is like putting all your ingredients in a basket before you start cooking.  
2. Next, it tells the computer to find the average price of all the items in the list. This is like calculating the total cost of all the ingredients and dividing it by the number of items.  
3. Then, it asks the computer to take the average price and multiply it by a number called "5." This is like adding a special seasoning to your recipe that makes everything taste better.  
4. Finally, it tells the computer to print out the result, which is like taking a bite of your delicious dish and saying "Mmm, this tastes great!"  
  
So, in summary, the code takes some prices, finds their average, multiplies it by a special number, and then prints out the result. It's like a simple recipe for a computer program that makes something useful happen!