2.代码分析

2.1代码结构分析

2.1.1代码

import os

import libcst as cst

flask\_repo\_path = "./flask-main" # 用实际的 Flask 仓库路径替换

def analyze\_code(file\_path):

with open(file\_path, "r", encoding="utf-8") as file:

code = file.read()

module = cst.parse\_module(code)

return module

def extract\_classes\_functions\_imports(node):

classes = [c for c in node.body if isinstance(c, cst.ClassDef)]

functions = [f for f in node.body if isinstance(f, cst.FunctionDef)]

imports = [i for i in node.body if isinstance(i, cst.Import) or isinstance(i, cst.ImportFrom)]

return classes, functions, imports

def explore\_module\_structure(module):

classes, functions, imports = extract\_classes\_functions\_imports(module)

print(f"Classes in module: {len(classes)}")

for class\_node in classes:

print(f" - {class\_node.name}")

print(f"\nFunctions in module: {len(functions)}")

for function\_node in functions:

print(f" - {function\_node.name}")

print(f"\nImports in module: {len(imports)}")

for import\_node in imports:

print(f" - {import\_node}")

def main():

for root, dirs, files in os.walk(flask\_repo\_path):

for file in files:

if file.endswith(".py"):

file\_path = os.path.join(root, file)

module = analyze\_code(file\_path)

print(f"\nAnalyzing module: {file\_path}")

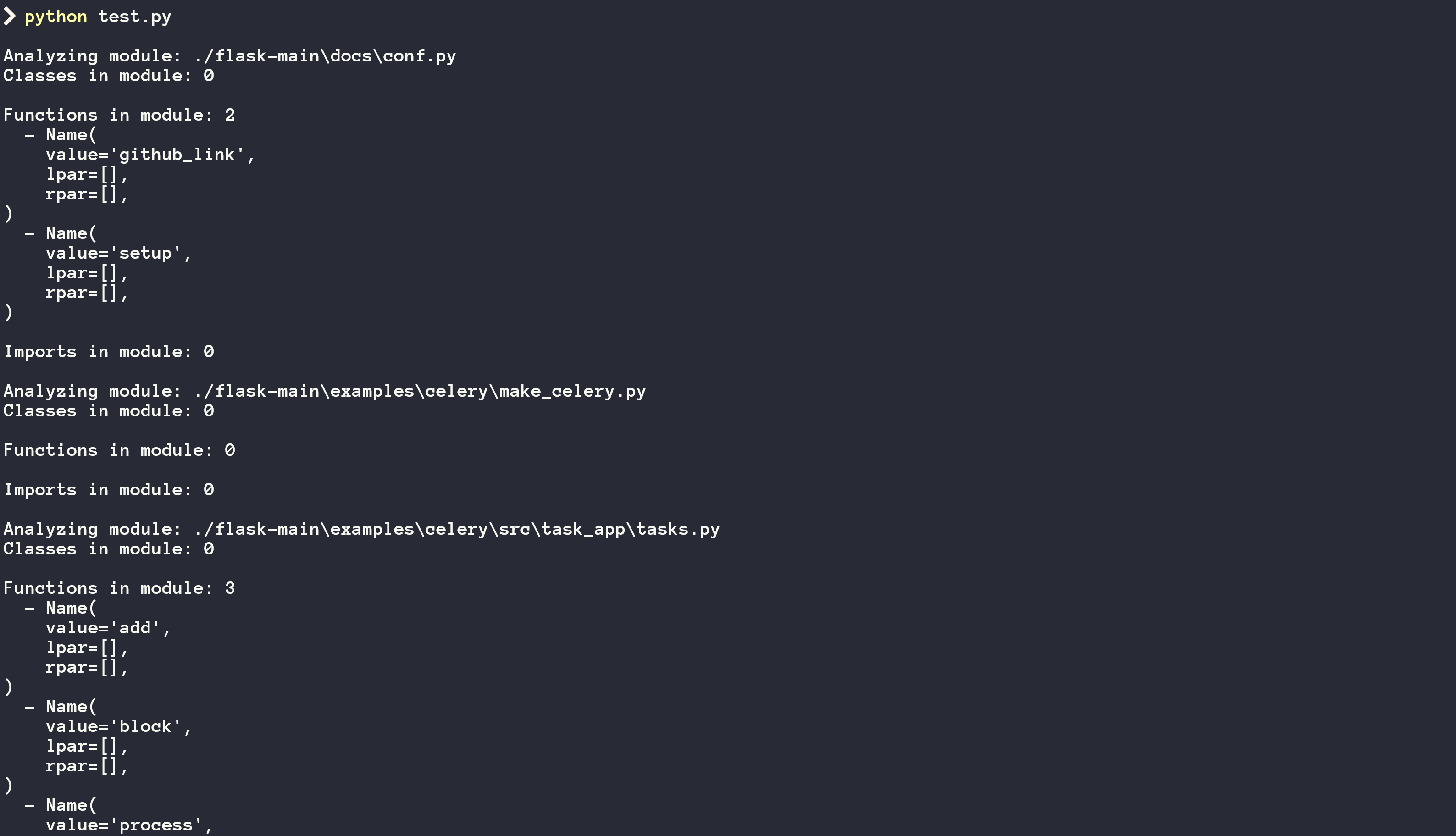
explore\_module\_structure(module)

if \_\_name\_\_ == "\_\_main\_\_":

main()

利用上述代码分析出每个python文件中的类、函数和导入语句

运行结果：



修改代码：

import os

import libcst as cst

flask\_repo\_path = "./flask-main" # 用实际的 Flask 仓库路径替换

output\_file\_path = "./output\_1.txt" # 指定输出文件路径

def analyze\_code(file\_path):

with open(file\_path, "r", encoding="utf-8") as file:

code = file.read()

module = cst.parse\_module(code)

return module

def extract\_classes\_functions\_imports(node):

classes = [c for c in node.body if isinstance(c, cst.ClassDef)]

functions = [f for f in node.body if isinstance(f, cst.FunctionDef)]

imports = [i for i in node.body if isinstance(i, cst.Import) or isinstance(i, cst.ImportFrom)]

return classes, functions, imports

def explore\_module\_structure(module, output\_file):

classes, functions, imports = extract\_classes\_functions\_imports(module)

output\_file.write(f"Classes in module: {len(classes)}\n")

for class\_node in classes:

output\_file.write(f" - {class\_node.name}\n")

output\_file.write(f"\nFunctions in module: {len(functions)}\n")

for function\_node in functions:

output\_file.write(f" - {function\_node.name}\n")

output\_file.write(f"\nImports in module: {len(imports)}\n")

for import\_node in imports:

output\_file.write(f" - {import\_node}\n")

def main():

with open(output\_file\_path, "w", encoding="utf-8") as output\_file:

for root, dirs, files in os.walk(flask\_repo\_path):

for file in files:

if file.endswith(".py"):

file\_path = os.path.join(root, file)

module = analyze\_code(file\_path)

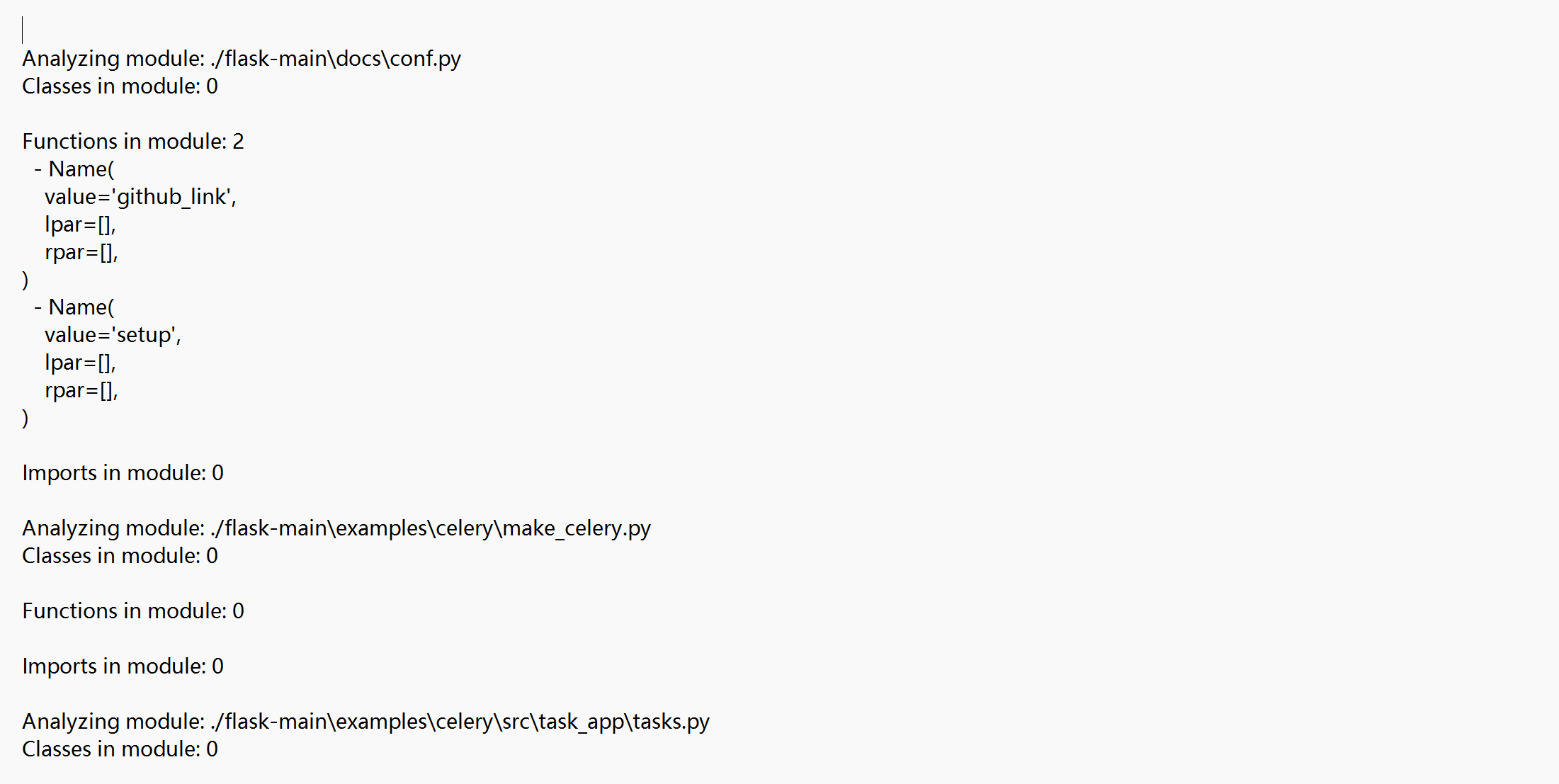
output\_file.write(f"\nAnalyzing module: {file\_path}\n")

explore\_module\_structure(module, output\_file)

if \_\_name\_\_ == "\_\_main\_\_":

main()

输出结果保存在txt中：



2.2代码函数调用分析

2.2.1代码

from pathlib import Path

from libcst import parse\_module, CSTNode, CSTVisitor, Call

# 定义一个访问器来查找特定模式

class CustomAnalyzer(CSTVisitor):

def \_\_init\_\_(self):

super().\_\_init\_\_() # 调用父类的构造函数

self.function\_calls = []

def visit\_Call(self, node: "Call") -> None:

# 检查是否是函数调用

if isinstance(node.func, CSTNode):

self.function\_calls.append(node.func)

super().visit\_Call(node) # 调用父类的方法

def analyze\_flask\_codebase(flask\_repo\_path):

# 遍历 Flask 代码库中的所有 Python 文件

for python\_file in flask\_repo\_path.glob("\*\*/\*.py"):

with python\_file.open("r", encoding="utf-8") as file:

try:

# 尝试解析当前文件

module = parse\_module(file.read())

# 初始化自定义分析器

analyzer = CustomAnalyzer()

module.visit(analyzer)

# 输出分析结果

print(f"File analyzed: {python\_file}")

print(f"Function calls found: {analyzer.function\_calls}")

except Exception as e:

print(f"Error analyzing {python\_file}: {e}")

def main():

# Flask 代码库的根目录

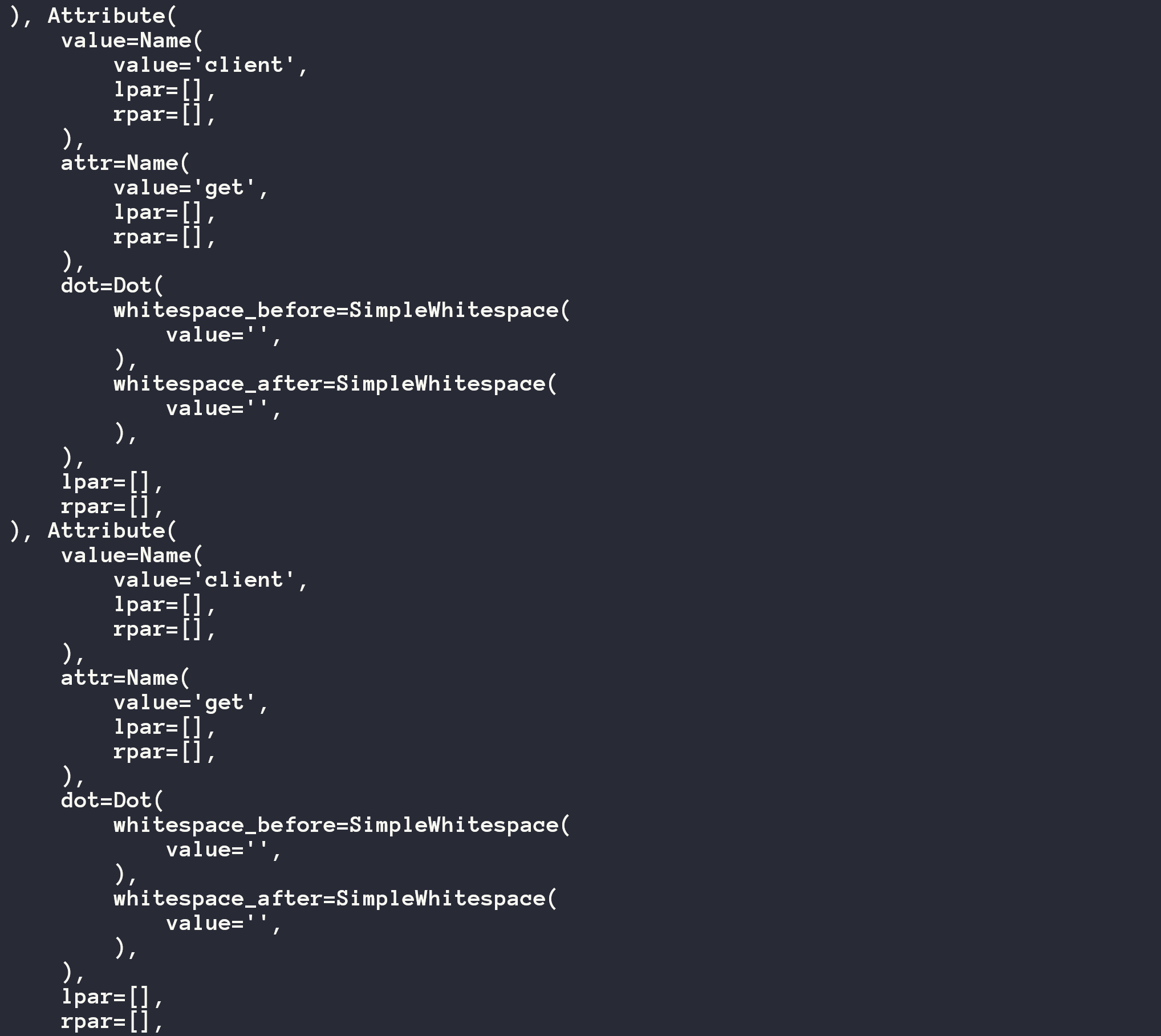
flask\_repo\_path = Path("./flask-main")

analyze\_flask\_codebase(flask\_repo\_path)

if \_\_name\_\_ == "\_\_main\_\_":

main()

运行结果：



修改代码使其将结果输出到txt文本中：

from pathlib import Path

from libcst import parse\_module, CSTNode, CSTVisitor, Call

# 定义一个访问器来查找特定模式

class CustomAnalyzer(CSTVisitor):

def \_\_init\_\_(self):

super().\_\_init\_\_() # 调用父类的构造函数

self.function\_calls = []

def visit\_Call(self, node: "Call") -> None:

# 检查是否是函数调用

if isinstance(node.func, CSTNode):

self.function\_calls.append(node.func)

super().visit\_Call(node) # 调用父类的方法

def analyze\_flask\_codebase(flask\_repo\_path, output\_file):

with open(output\_file, 'w', encoding='utf-8') as output:

# 遍历 Flask 代码库中的所有 Python 文件

for python\_file in flask\_repo\_path.glob("\*\*/\*.py"):

with python\_file.open("r", encoding="utf-8") as file:

try:

# 尝试解析当前文件

module = parse\_module(file.read())

# 初始化自定义分析器

analyzer = CustomAnalyzer()

module.visit(analyzer)

# 输出分析结果到文件

output.write(f"File analyzed: {python\_file}\n")

output.write(f"Function calls found: {analyzer.function\_calls}\n")

output.write("\n")

except Exception as e:

output.write(f"Error analyzing {python\_file}: {e}\n")

def main():

# Flask 代码库的根目录

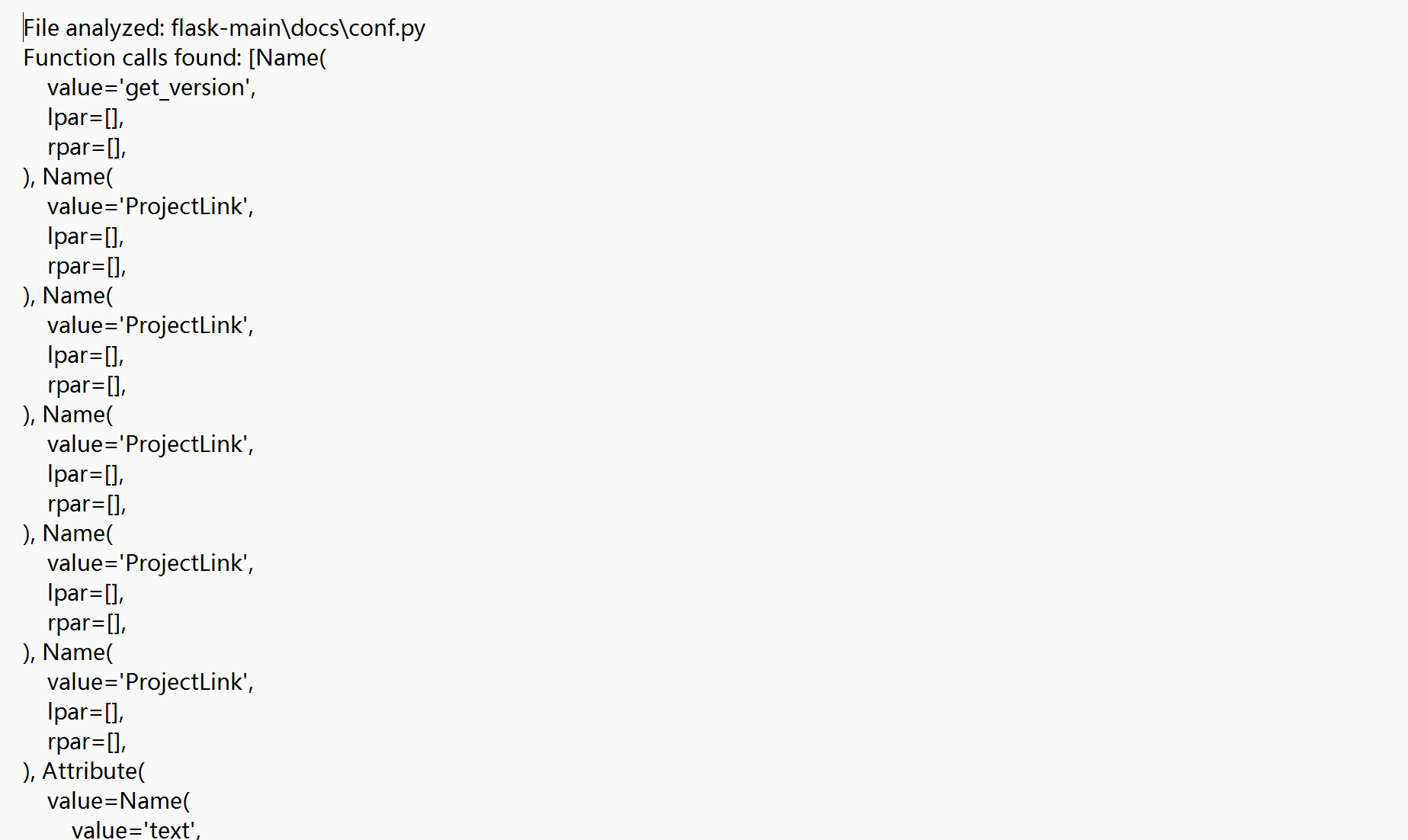
flask\_repo\_path = Path("./flask-main")

output\_file = "call\_result.txt"

analyze\_flask\_codebase(flask\_repo\_path, output\_file)

if \_\_name\_\_ == "\_\_main\_\_":

main()



2.3代码复杂度（以函数个数为标准）

2.3.1代码：

from pathlib import Path

from libcst import parse\_module, CSTNode, CSTVisitor, Call, FunctionDef

# 定义一个访问器来查找特定模式

class CustomAnalyzer(CSTVisitor):

def \_\_init\_\_(self):

super().\_\_init\_\_() # 调用父类的构造函数

self.function\_calls = []

self.function\_count = 0

def visit\_Call(self, node: "Call") -> None:

# 检查是否是函数调用

if isinstance(node.func, CSTNode):

self.function\_calls.append(node.func)

super().visit\_Call(node) # 调用父类的方法

def visit\_FunctionDef(self, node: "FunctionDef") -> None:

# 统计函数数量

self.function\_count += 1

super().visit\_FunctionDef(node)

def analyze\_flask\_codebase(flask\_repo\_path, output\_file):

with open(output\_file, 'w', encoding='utf-8') as output:

# 遍历 Flask 代码库中的所有 Python 文件

for python\_file in flask\_repo\_path.glob("\*\*/\*.py"):

with python\_file.open("r", encoding="utf-8") as file:

try:

# 尝试解析当前文件

module = parse\_module(file.read())

# 初始化自定义分析器

analyzer = CustomAnalyzer()

module.visit(analyzer)

# 输出分析结果到文件

output.write(f"File analyzed: {python\_file}\n")

output.write(f"Function calls found: {analyzer.function\_calls}\n")

output.write(f"Function count: {analyzer.function\_count}\n")

output.write("\n")

except Exception as e:

output.write(f"Error analyzing {python\_file}: {e}\n")

def main():

# Flask 代码库的根目录

flask\_repo\_path = Path("./flask-main")

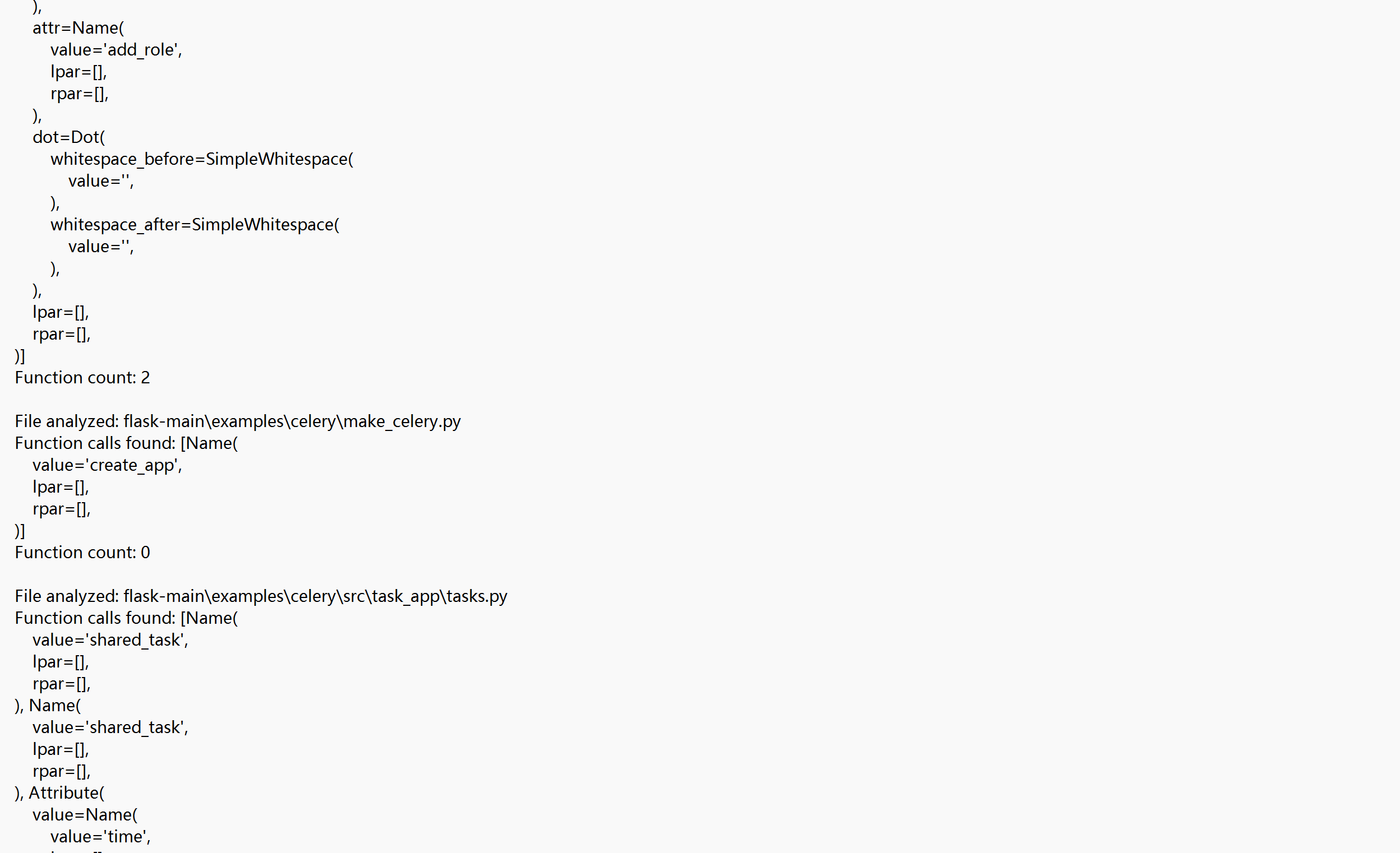
output\_file = "complex\_results.txt"

analyze\_flask\_codebase(flask\_repo\_path, output\_file)

if \_\_name\_\_ == "\_\_main\_\_":

main()

输出结果将保存在txt文件中



5.bug修复

5.1 使用git log命令得到所有提交记录

5.2 使用git log --grep="bug\|fix" >> log\_bugfix.txt将关于修复bug的提交记录检索出来并保存到文件中

5.2.1查看每次提交所作改变

5.2.1.1 代码

import subprocess

# Read the log file

with open('./log\_bugfix.txt', 'r',encoding='utf-16 le') as log\_file:

log\_content = log\_file.read()

# Split the log content into individual commits

commits = log\_content.split('\n\n')

# Open a file for writing the output

output\_file\_name = 'output\_bugfix\_detail.txt'

with open(output\_file\_name, 'w') as output\_file:

# Iterate through each commit

for commit in commits:

# Split the commit into lines

commit\_lines = commit.split('\n', 1) # Split only at the first newline

# Extract the commit SHA value

commit\_info = commit\_lines[0].split()

if len(commit\_info) >= 2 and commit\_info[0] == 'commit':

commit\_sha = commit\_info[1]

print(f'Processing commit {commit\_sha}')

# Use git show to get commit information

git\_show\_output = subprocess.check\_output(['git', 'show', commit\_sha], universal\_newlines=True)

# Write the git show output to the output file

output\_file.write(f'Git show output for commit {commit\_sha}:\n')

output\_file.write(git\_show\_output)

output\_file.write('\n' + '-' \* 80 + '\n') # Separating commits

else:

print(f'Skipping invalid commit: {commit\_lines[0]}')

print(f'Git show outputs written to {output\_file\_name}')

结果将输出到output\_bugfix\_detail.txt文件中

5.3 使用git log --grep="performance\|optimization" >> log\_performance.txt命令检索出关于性能提升的提交记录并保存

5.3.1查看提交所作改变

5.3.1.1 代码

git show 6979265fa643ed982d062f38d386c37bbbef0d9b >> output\_performance\_detail.txt

git show 6e52355eb303e1b47f7b83ddbb94ff432e2df139 >> output\_performance\_detail.txt

结果将保存在output\_performance\_detail.txt中。