## Overview  
  
This file provides utility functions for statically analyzing Python code. It uses Python's built-in `ast` (Abstract Syntax Tree) module to parse code content and extract structural information about its components, such as functions and classes, without executing the code.  
  
## Functions  
  
### `get\_functions\_and\_classes()`  
  
Parses a string of Python code to identify and extract metadata about all defined functions (synchronous and asynchronous) and classes.  
  
The function walks through the Abstract Syntax Tree (AST) of the provided code and collects details for each function and class definition it finds. If the input code contains syntax errors, the function will gracefully handle the exception and return an empty dictionary to prevent breaking the analysis process.  
  
#### Parameters  
  
- `code\_content (str)`: A string containing the Python source code to be analyzed.  
  
#### Returns  
  
- `Dict[str, Dict[str, Any]]`: A dictionary where each key is the name of a discovered function or class. The corresponding value is another dictionary containing the following metadata:  
 - `type (str)`: The type of the object, either "Function" or "Class".  
 - `name (str)`: The name of the function or class.  
 - `start\_line (int)`: The line number where the definition of the object begins.  
 - `end\_line (int)`: The line number where the definition of the object ends.  
  
#### Usage Example  
  
```python  
import ast  
from typing import List, Tuple, Dict, Any  
  
def get\_functions\_and\_classes(code\_content: str) -> Dict[str, Dict[str, Any]]:  
 """  
 Parses Python code content to extract information about functions and classes.  
 """  
 try:  
 tree = ast.parse(code\_content)  
 except SyntaxError:  
 # If file has syntax errors, return empty so it doesn't break analysis  
 return {}  
 objects = {}  
  
 for node in ast.walk(tree):  
 if isinstance(node, (ast.FunctionDef, ast.ClassDef, ast.AsyncFunctionDef)):  
 obj\_type = "Class" if isinstance(node, ast.ClassDef) else "Function"  
 objects[node.name] = {  
 "type": obj\_type,  
 "name": node.name,  
 "start\_line": node.lineno,  
 "end\_line": node.end\_lineno  
 }  
 return objects  
  
# Example usage:  
code = """  
import os  
  
class MyClass:  
 def \_\_init\_\_(self):  
 pass  
  
 def my\_method(self):  
 return "Hello"  
  
def my\_function():  
 print("World")  
"""  
  
analysis\_result = get\_functions\_and\_classes(code)  
import json  
print(json.dumps(analysis\_result, indent=2))  
  
# Expected Output:  
# {  
# "MyClass": {  
# "type": "Class",  
# "name": "MyClass",  
# "start\_line": 3,  
# "end\_line": 9  
# },  
# "my\_function": {  
# "type": "Function",  
# "name": "my\_function",  
# "start\_line": 11,  
# "end\_line": 12  
# },  
# "\_\_init\_\_": {  
# "type": "Function",  
# "name": "\_\_init\_\_",  
# "start\_line": 4,  
# "end\_line": 6  
# },  
# "my\_method": {  
# "type": "Function",  
# "name": "my\_method",  
# "start\_line": 8,  
# "end\_line": 9  
# }  
# }

<!-- DOC\_START: code\_monitor/utils.py::hello -->  
### `print(name)`  
  
This line of code outputs the value of the `name` variable to the standard output, which is typically the console or terminal.  
  
#### Functionality  
  
The statement utilizes Python's built-in `print()` function to display information during program execution.  
  
1. \*\*`print()` function\*\*: A standard Python function that sends data to the standard output stream (`sys.stdout`).  
2. \*\*`name` argument\*\*: The variable `name` is passed as an argument to the function. The `print()` function will display the string representation of the object that `name` refers to. This variable must be defined and assigned a value in the scope where this line is executed.  
3. \*\*Output\*\*: By default, `print()` appends a newline character (`\n`) to its output, causing subsequent output to appear on a new line.  
  
#### Context and Use Case  
  
This type of statement is frequently used for:  
  
\* \*\*Debugging\*\*: To quickly inspect the value of a variable at a specific point in the code's execution.  
\* \*\*Informational Output\*\*: To provide status updates, results, or simple logging in command-line applications.  
  
While useful for development, in production or more complex applications, it is often better to use a dedicated logging framework (like Python's `logging` module) for more controlled and configurable output.  
  
#### Example  
  
```python  
# Assume 'name' is defined earlier in the code  
name = "Code Analysis Task"  
  
# The line in question will print the variable's value to the console  
print(name)

**Console Output:**

Code Analysis Task

### fibonacci(n)

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

This function computes the value at a specific position n in the Fibonacci sequence, which starts with 0 and 1. It handles the initial base cases and then iteratively builds the sequence by summing the two preceding numbers until it reaches the desired position.

* If n is 0 or a negative number, the function returns 0.
* If n is 1, it returns 1.
* For n greater than 1, it loops n-1 times to compute the final value.

#### Parameters

* n (int): The position in the Fibonacci sequence. It is expected to be a non-negative integer.

#### Returns

* int: The n-th Fibonacci number.

#### Example

# Calculate the 10th Fibonacci number  
tenth\_fib = fibonacci(10)  
print(tenth\_fib)  
# Expected Output: 55  
  
# Handle base cases  
zero\_fib = fibonacci(0)  
print(zero\_fib)  
# Expected Output: 0