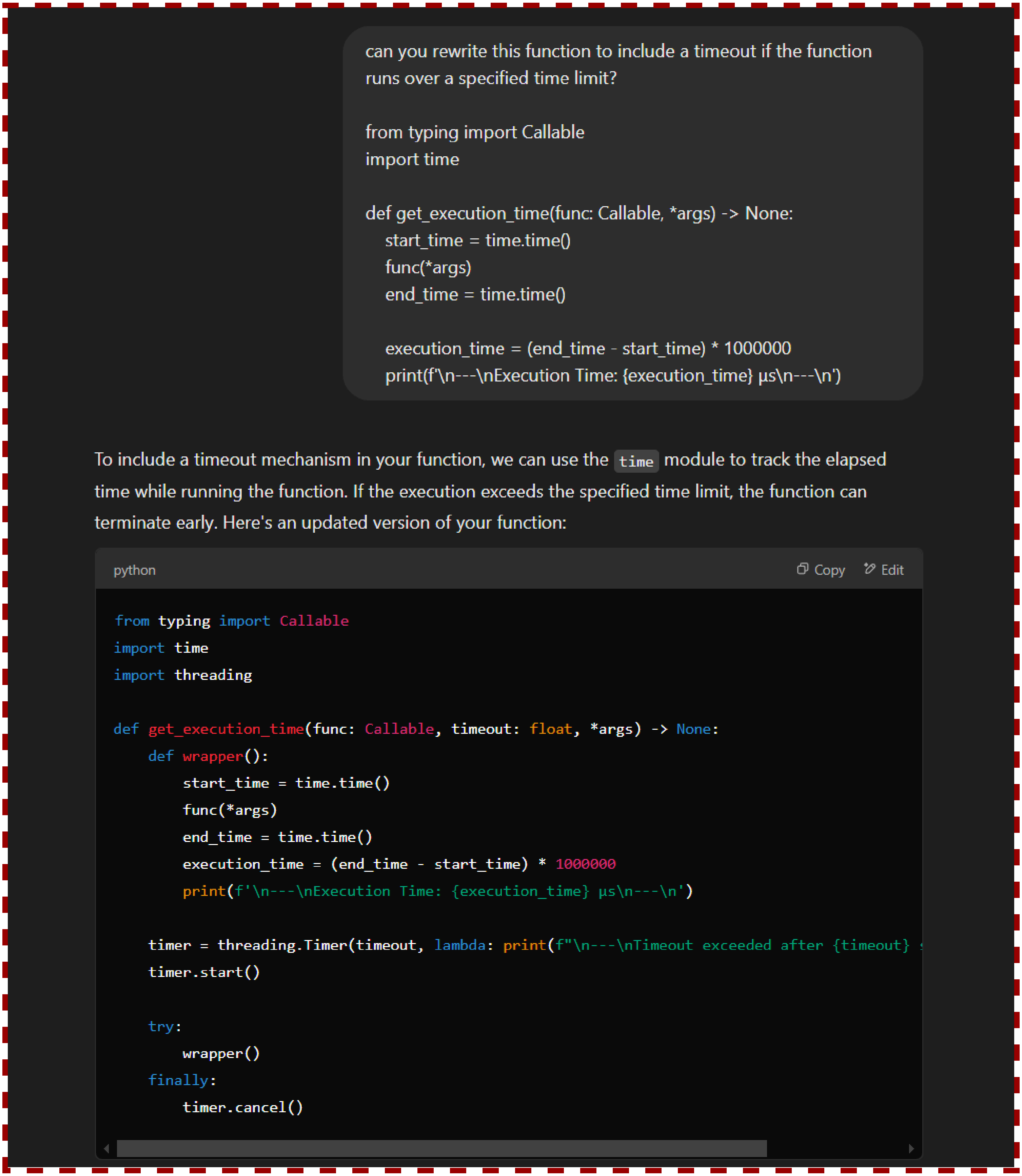
Timeout Functionality

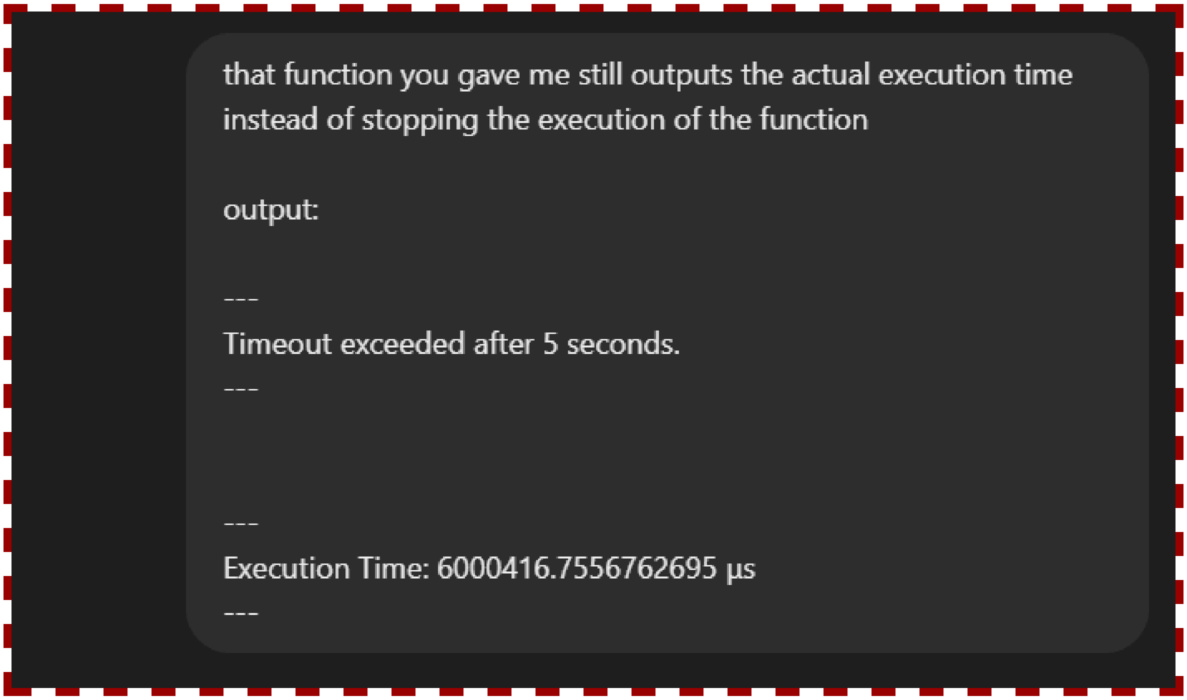


can you rewrite this function to include a timeout if the function runs over a specified time limit?

```
from typing import Callable
import time

def get_execution_time(func: Callable, *args) -> None:
    start_time = time.time()
    func(*args)
    end_time = time.time()

    execution_time = (end_time - start_time) * 1000000
    print(f'\n---\nExecution Time: {execution_time} µs\n---\n')
```

To include a timeout mechanism in your function, we can use the `time` module to track the elapsed time while running the function. If the execution exceeds the specified time limit, the function can terminate early. Here's an updated version of your function:

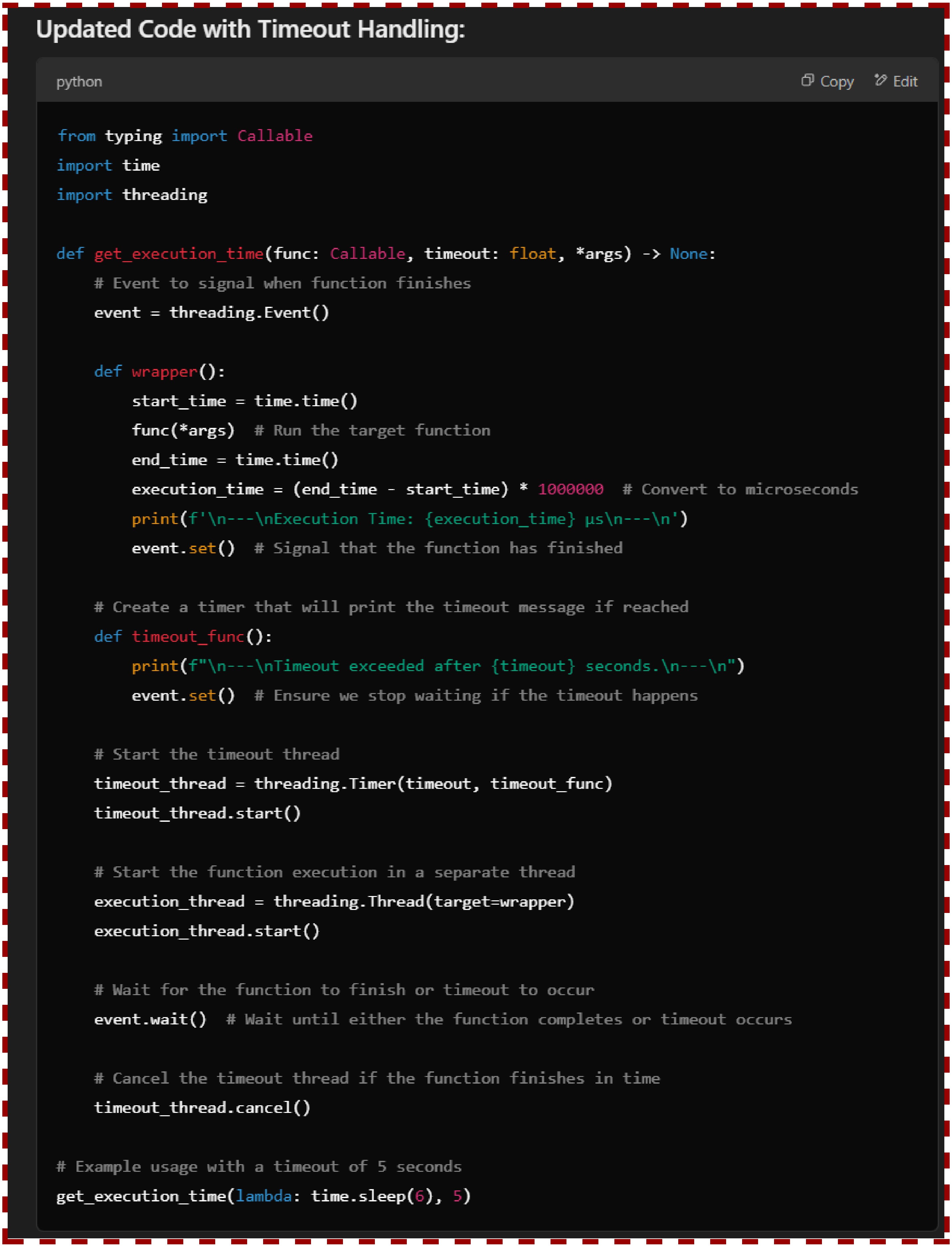
```python
from typing import Callable
import time
import threading

def get_execution_time(func: Callable, timeout: float, *args) -> None:
    def wrapper():
        start_time = time.time()
        func(*args)
        end_time = time.time()
        execution_time = (end_time - start_time) * 1000000
        print(f'\n---\nExecution Time: {execution_time} µs\n---\n')

    timer = threading.Timer(timeout, lambda: print(f"\n---\nTimeout exceeded after {timeout}
    timer.start()

    try:
        wrapper()
    finally:
        timer.cancel()
```



that function you gave me still outputs the actual execution time instead of stopping the execution of the function

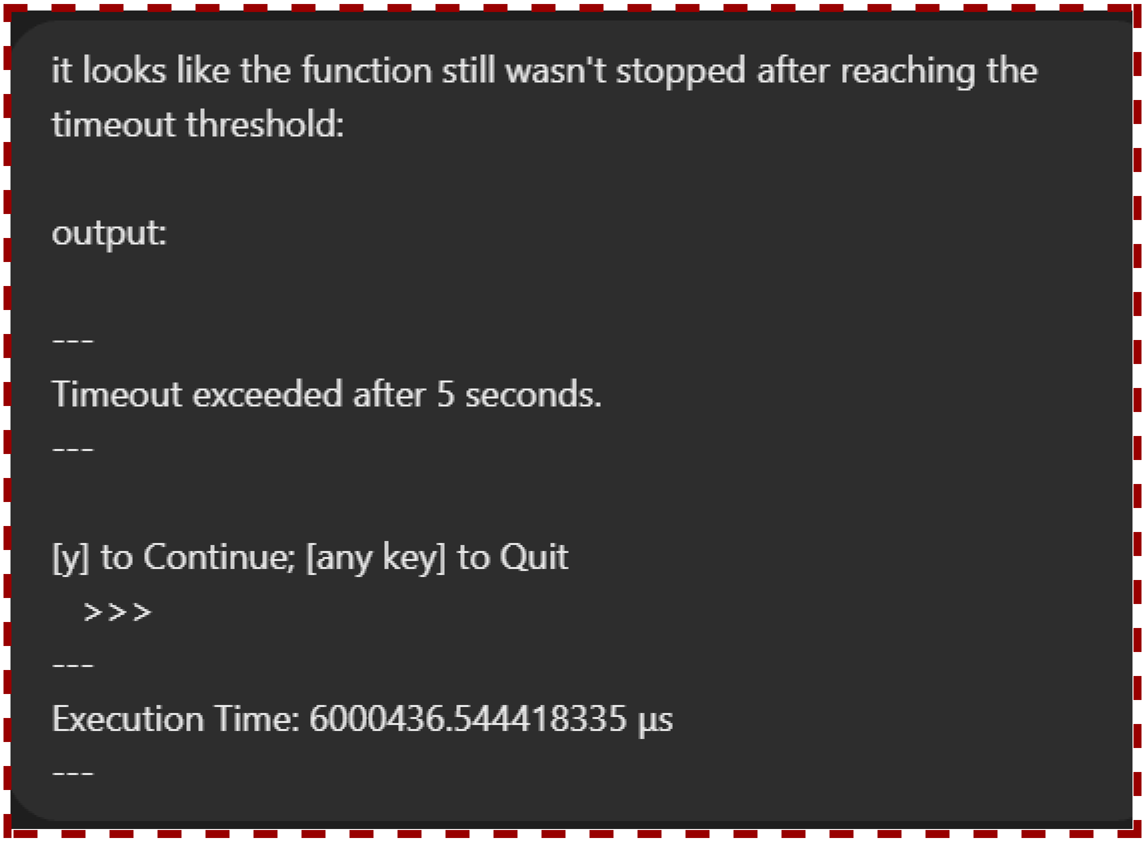
output:

```
---
Timeout exceeded after 5 seconds.
---



---
Execution Time: 6000416.7556762695 μs
---
```



Updated Code with Timeout Handling:

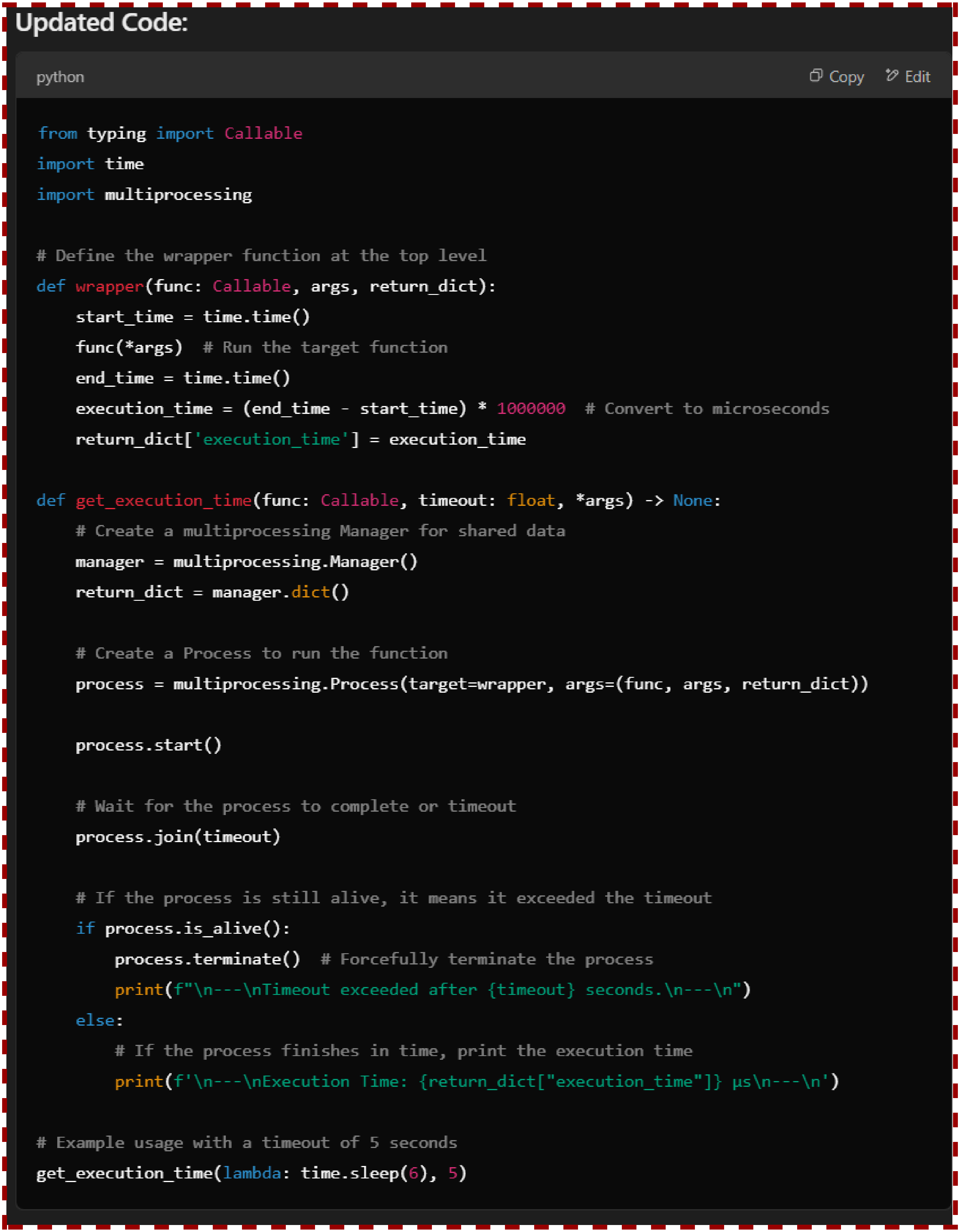
```python
from typing import Callable
import time
import threading

def get_execution_time(func: Callable, timeout: float, *args) -> None:
    # Event to signal when function finishes
    event = threading.Event()

    def wrapper():
        start_time = time.time()
        func(*args)  # Run the target function
        end_time = time.time()
        execution_time = (end_time - start_time) * 1000000  # Convert to microseconds
        print(f'\n---\nExecution Time: {execution_time} µs\n---\n')
        event.set()  # Signal that the function has finished

    # Create a timer that will print the timeout message if reached
    def timeout_func():
        print(f"\n---\nTimeout exceeded after {timeout} seconds.\n---\n")
        event.set()  # Ensure we stop waiting if the timeout happens

    # Start the timeout thread
    timeout_thread = threading.Timer(timeout, timeout_func)
    timeout_thread.start()

    # Start the function execution in a separate thread
    execution_thread = threading.Thread(target=wrapper)
    execution_thread.start()

    # Wait for the function to finish or timeout to occur
    event.wait()  # Wait until either the function completes or timeout occurs

    # Cancel the timeout thread if the function finishes in time
    timeout_thread.cancel()

# Example usage with a timeout of 5 seconds
get_execution_time(lambda: time.sleep(6), 5)
```



it looks like the function still wasn't stopped after reaching the timeout threshold:

output:

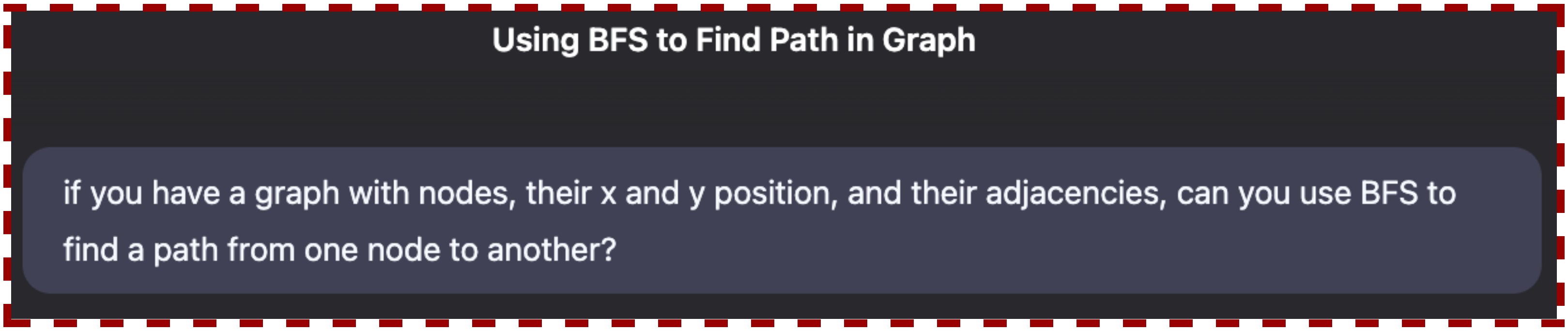
```
---
Timeout exceeded after 5 seconds.
---

[y] to Continue; [any key] to Quit
   >>>
---
Execution Time: 6000436.544418335 µs
---
```



Updated Code:

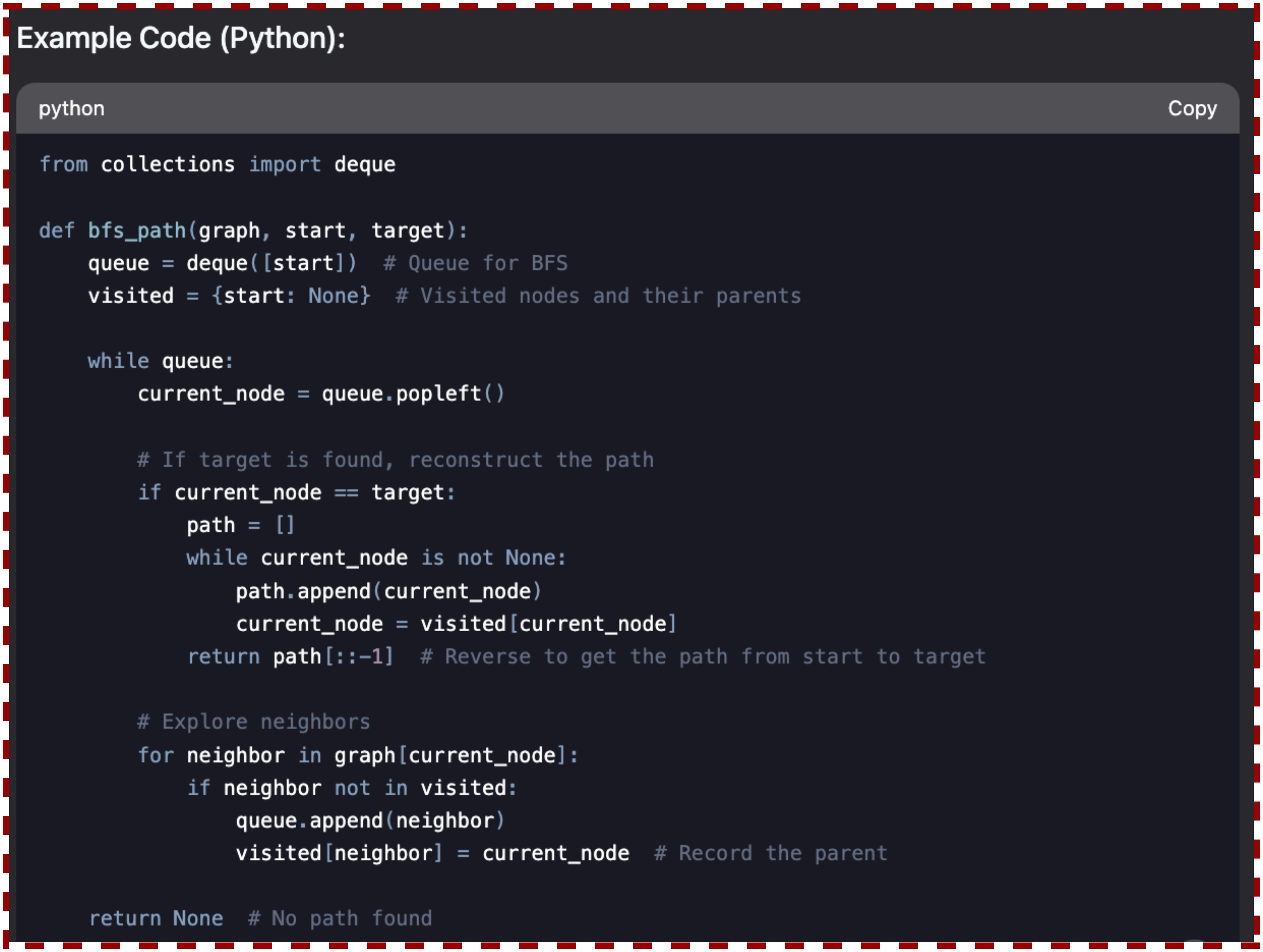
```python
from typing import Callable
import time
import multiprocessing

# Define the wrapper function at the top level
def wrapper(func: Callable, args, return_dict):
    start_time = time.time()
    func(*args)  # Run the target function
    end_time = time.time()
    execution_time = (end_time - start_time) * 1000000  # Convert to microseconds
    return_dict['execution_time'] = execution_time

def get_execution_time(func: Callable, timeout: float, *args) -> None:
    # Create a multiprocessing Manager for shared data
    manager = multiprocessing.Manager()
    return_dict = manager.dict()

    # Create a Process to run the function
    process = multiprocessing.Process(target=wrapper, args=(func, args, return_dict))

    process.start()

    # Wait for the process to complete or timeout
    process.join(timeout)

    # If the process is still alive, it means it exceeded the timeout
    if process.is_alive():
        process.terminate()  # Forcefully terminate the process
        print(f"\n---\nTimeout exceeded after {timeout} seconds.\n---\n")
    else:
        # If the process finishes in time, print the execution time
        print(f'\n---\nExecution Time: {return_dict["execution_time"]} µs\n---\n')

# Example usage with a timeout of 5 seconds
get_execution_time(lambda: time.sleep(6), 5)
```

BFS



Using BFS to Find Path in Graph

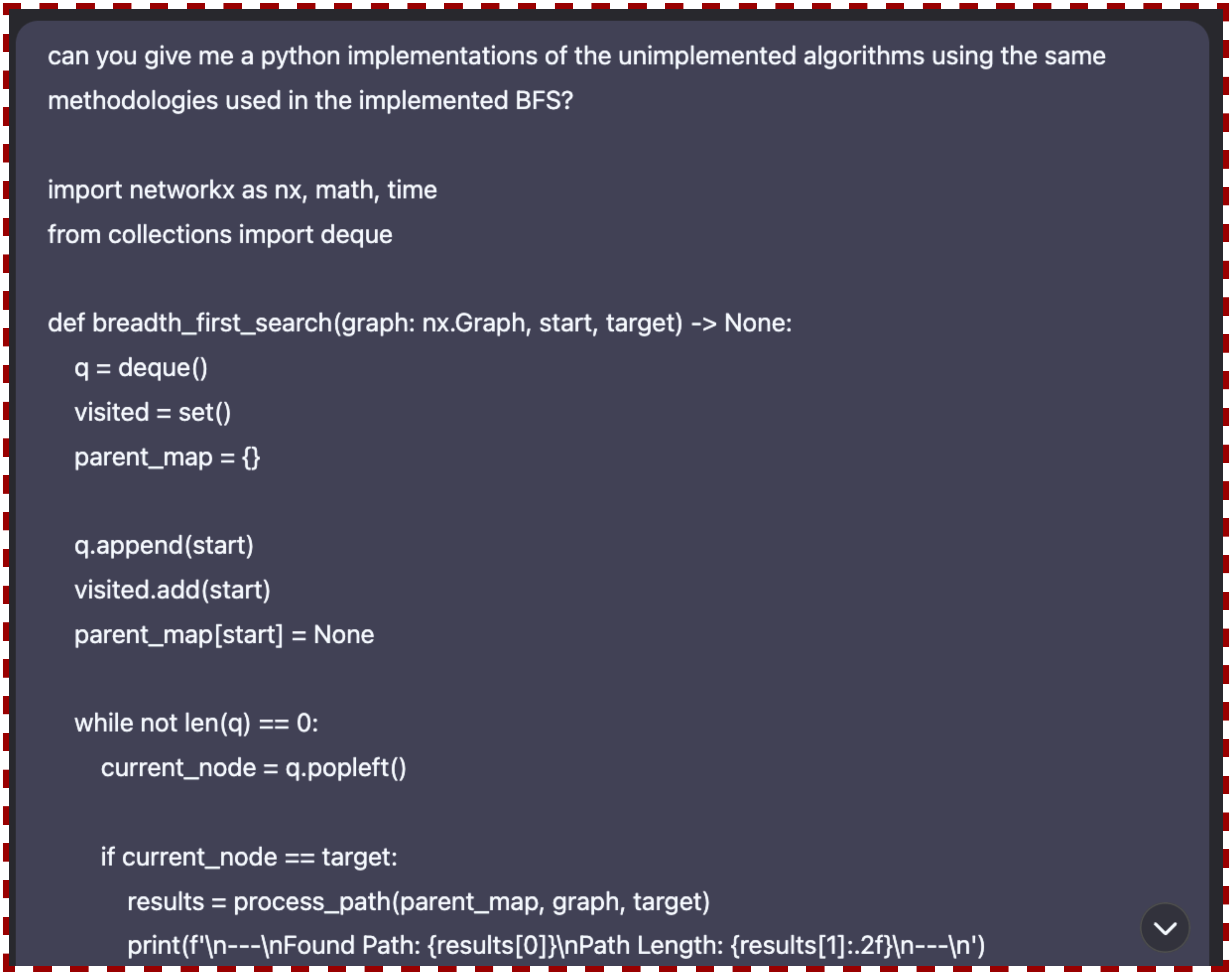
if you have a graph with nodes, their x and y position, and their adjacencies, can you use BFS to find a path from one node to another?



Example Code (Python):

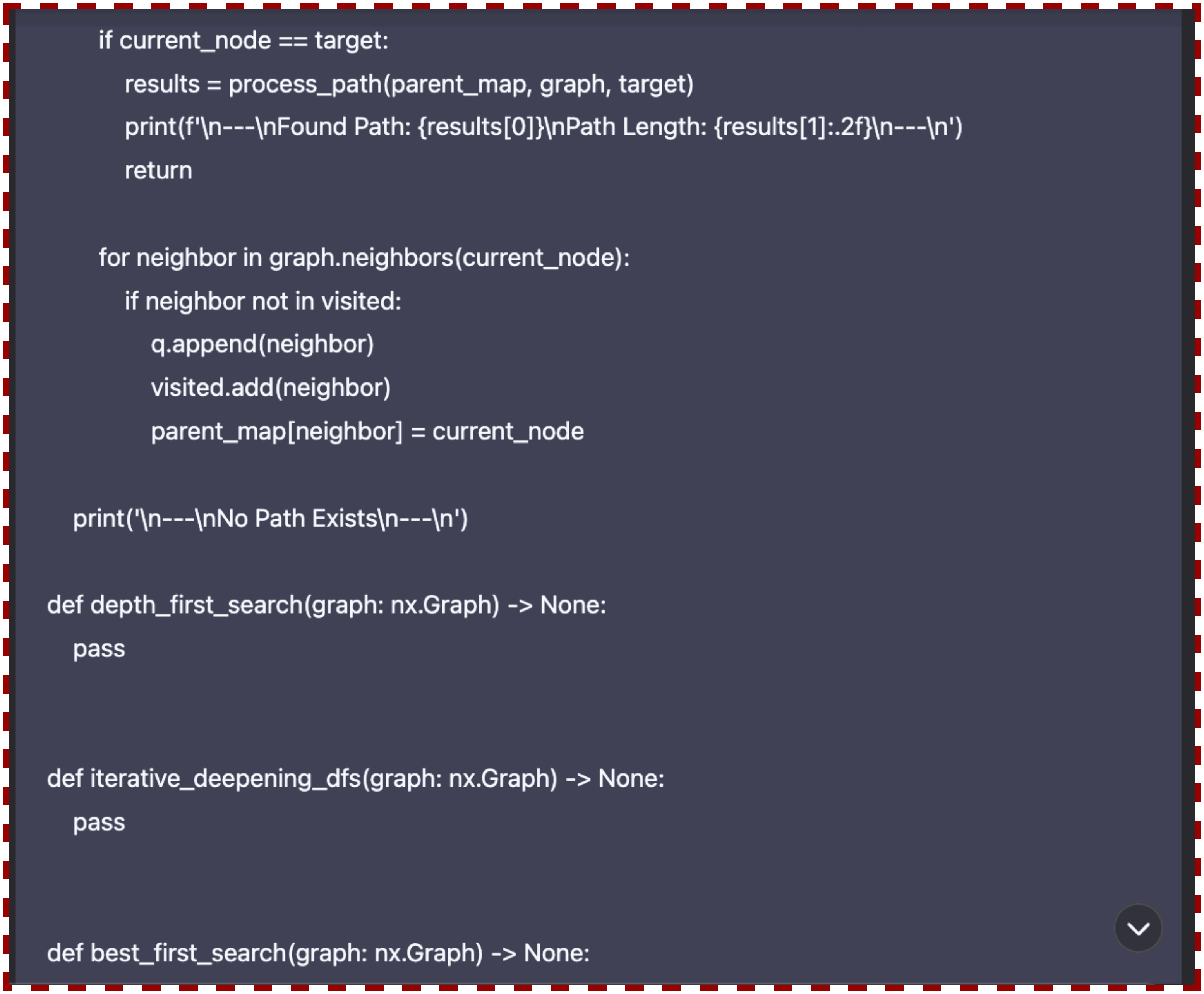
```python
from collections import deque

def bfs_path(graph, start, target):
    queue = deque([start])  # Queue for BFS
    visited = {start: None}  # Visited nodes and their parents

    while queue:
        current_node = queue.popleft()

        # If target is found, reconstruct the path
        if current_node == target:
            path = []
            while current_node is not None:
                path.append(current_node)
                current_node = visited[current_node]
            return path[::-1]  # Reverse to get the path from start to target

        # Explore neighbors
        for neighbor in graph[current_node]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited[neighbor] = current_node  # Record the parent

    return None  # No path found
```

All Other Algorithms

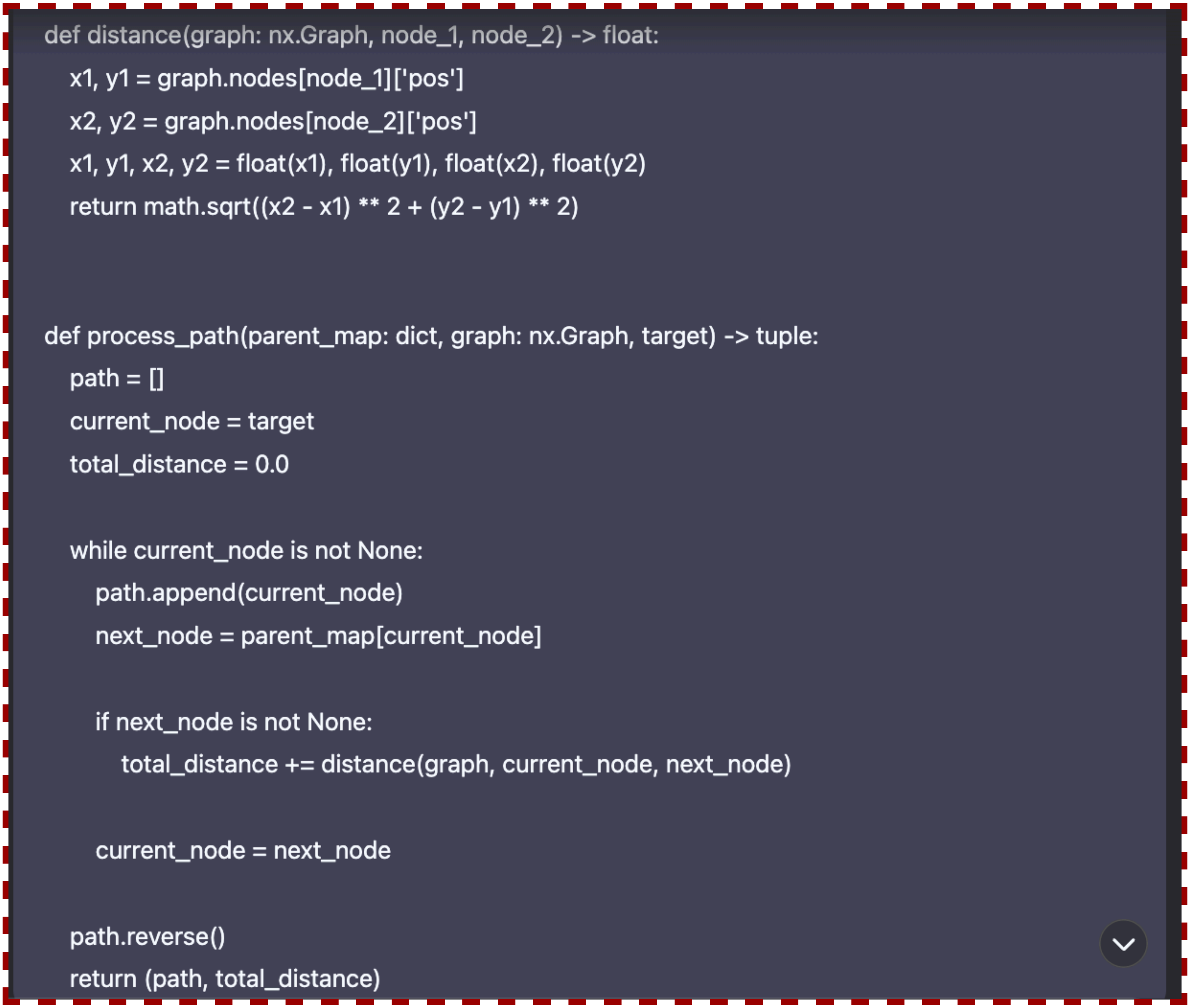


can you give me a python implementations of the unimplemented algorithms using the same methodologies used in the implemented BFS?

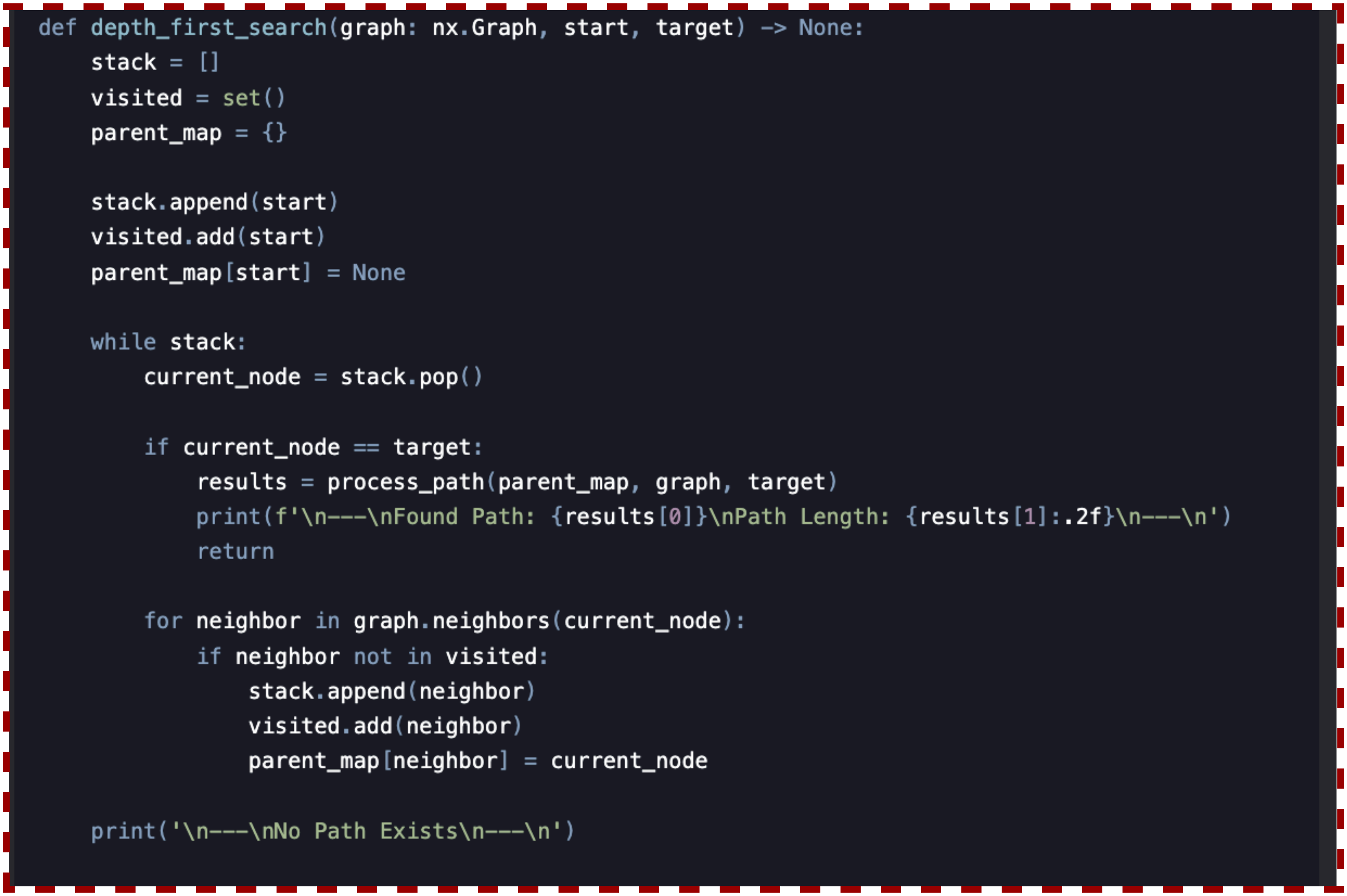
```
import networkx as nx, math, time
from collections import deque

def breadth_first_search(graph: nx.Graph, start, target) -> None:
    q = deque()
    visited = set()
    parent_map = {}

    q.append(start)
    visited.add(start)
    parent_map[start] = None

    while not len(q) == 0:
        current_node = q.popleft()

        if current_node == target:
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
```



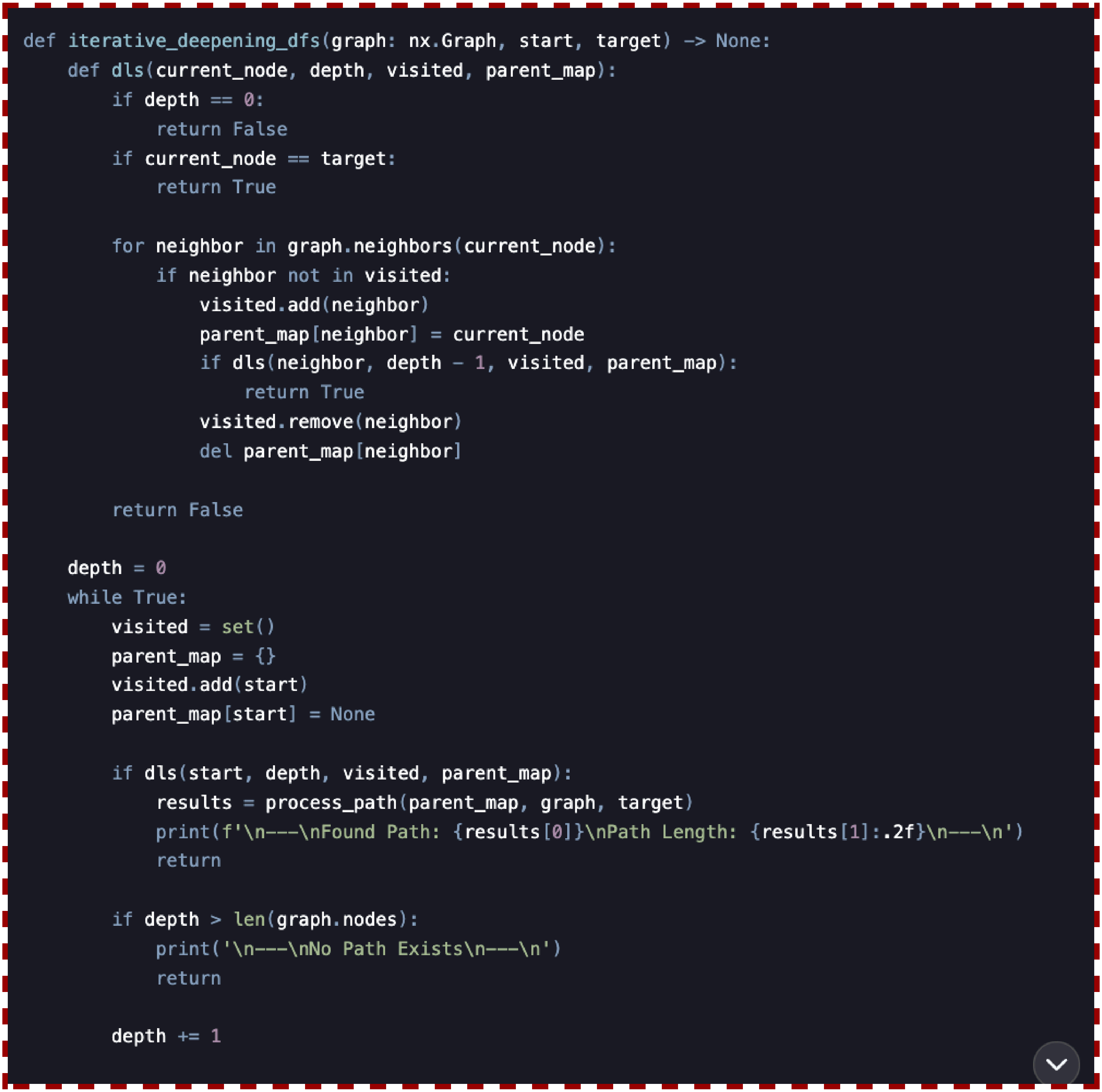
```
        if current_node == target:
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
            return

        for neighbor in graph.neighbors(current_node):
            if neighbor not in visited:
                q.append(neighbor)
                visited.add(neighbor)
                parent_map[neighbor] = current_node

    print('\n---\nNo Path Exists\n---\n')

def depth_first_search(graph: nx.Graph) -> None:
    pass


def iterative_deepening_dfs(graph: nx.Graph) -> None:
    pass


def best_first_search(graph: nx.Graph) -> None:
```



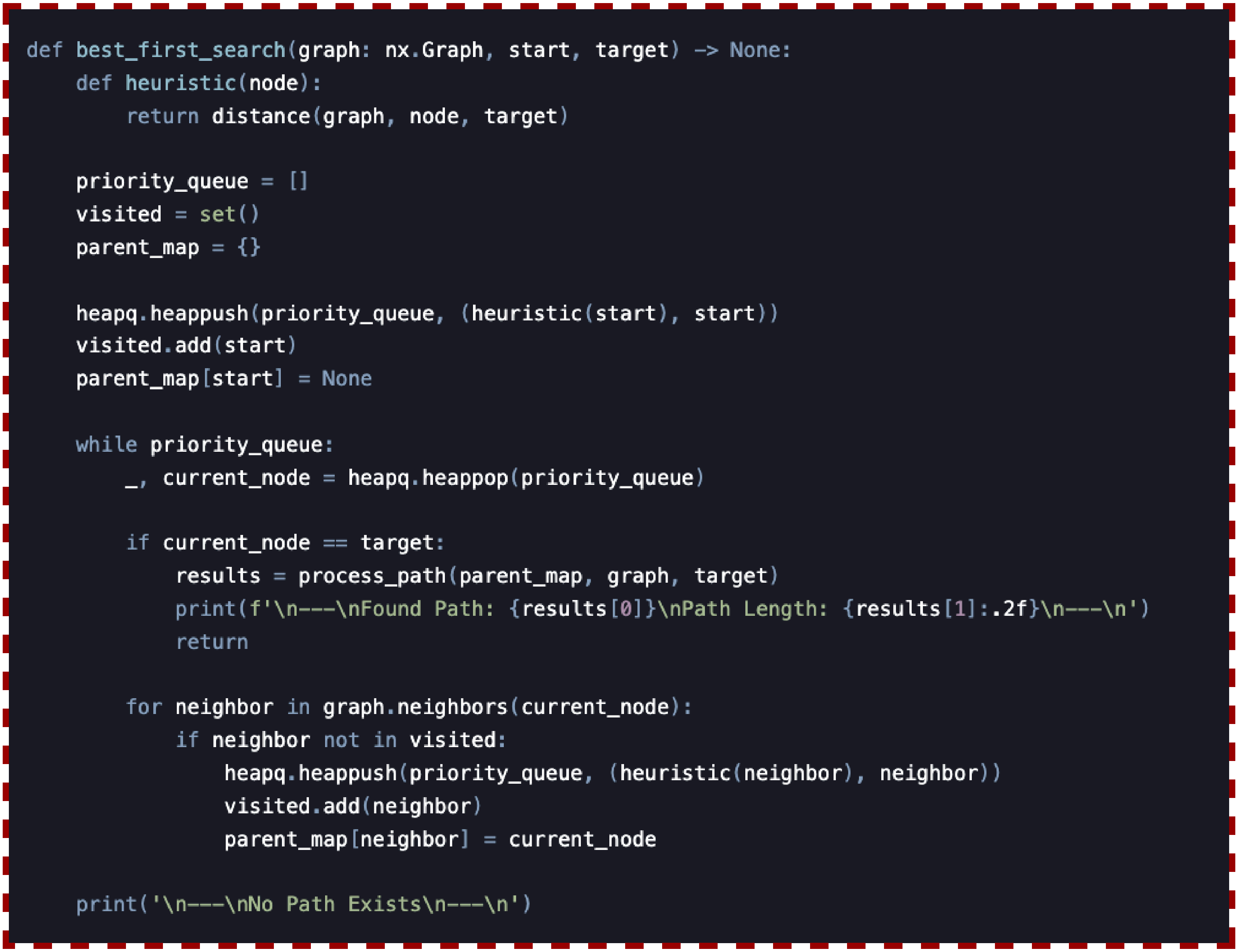
```
def distance(graph: nx.Graph, node_1, node_2) -> float:
    x1, y1 = graph.nodes[node_1]['pos']
    x2, y2 = graph.nodes[node_2]['pos']
    x1, y1, x2, y2 = float(x1), float(y1), float(x2), float(y2)
    return math.sqrt((x2 - x1) ** 2 + (y2 - y1) ** 2)



def process_path(parent_map: dict, graph: nx.Graph, target) -> tuple:
    path = []
    current_node = target
    total_distance = 0.0

    while current_node is not None:
        path.append(current_node)
        next_node = parent_map[current_node]

        if next_node is not None:
            total_distance += distance(graph, current_node, next_node)

        current_node = next_node

    path.reverse()
    return (path, total_distance)
```



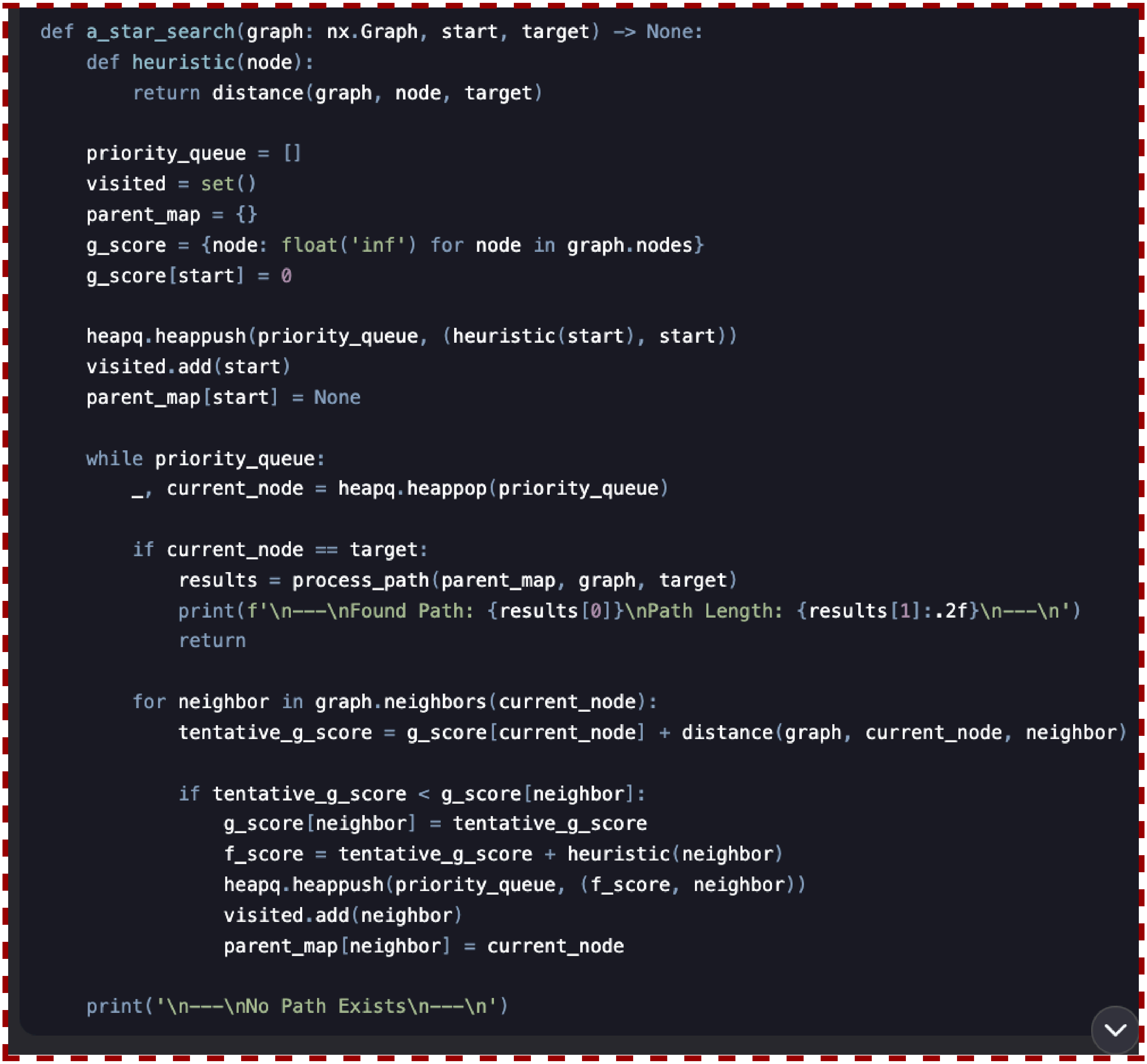
```python
def depth_first_search(graph: nx.Graph, start, target) -> None:
    stack = []
    visited = set()
    parent_map = {}

    stack.append(start)
    visited.add(start)
    parent_map[start] = None

    while stack:
        current_node = stack.pop()

        if current_node == target:
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
            return

        for neighbor in graph.neighbors(current_node):
            if neighbor not in visited:
                stack.append(neighbor)
                visited.add(neighbor)
                parent_map[neighbor] = current_node

    print('\n---\nNo Path Exists\n---\n')
```



```
def iterative_deepening_dfs(graph: nx.Graph, start, target) -> None:
    def dls(current_node, depth, visited, parent_map):
        if depth == 0:
            return False
        if current_node == target:
            return True

        for neighbor in graph.neighbors(current_node):
            if neighbor not in visited:
                visited.add(neighbor)
                parent_map[neighbor] = current_node
                if dls(neighbor, depth - 1, visited, parent_map):
                    return True
                visited.remove(neighbor)
                del parent_map[neighbor]

        return False

    depth = 0
    while True:
        visited = set()
        parent_map = {}
        visited.add(start)
        parent_map[start] = None

        if dls(start, depth, visited, parent_map):
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
            return

        if depth > len(graph.nodes):
            print('\n---\nNo Path Exists\n---\n')
            return

        depth += 1
```



```python
def best_first_search(graph: nx.Graph, start, target) -> None:
    def heuristic(node):
        return distance(graph, node, target)

    priority_queue = []
    visited = set()
    parent_map = {}

    heapq.heappush(priority_queue, (heuristic(start), start))
    visited.add(start)
    parent_map[start] = None

    while priority_queue:
        _, current_node = heapq.heappop(priority_queue)

        if current_node == target:
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
            return

        for neighbor in graph.neighbors(current_node):
            if neighbor not in visited:
                heapq.heappush(priority_queue, (heuristic(neighbor), neighbor))
                visited.add(neighbor)
                parent_map[neighbor] = current_node

    print('\n---\nNo Path Exists\n---\n')
```



```
def a_star_search(graph: nx.Graph, start, target) -> None:
    def heuristic(node):
        return distance(graph, node, target)

    priority_queue = []
    visited = set()
    parent_map = {}
    g_score = {node: float('inf') for node in graph.nodes}
    g_score[start] = 0

    heapq.heappush(priority_queue, (heuristic(start), start))
    visited.add(start)
    parent_map[start] = None

    while priority_queue:
        _, current_node = heapq.heappop(priority_queue)

        if current_node == target:
            results = process_path(parent_map, graph, target)
            print(f'\n---\nFound Path: {results[0]}\nPath Length: {results[1]:.2f}\n---\n')
            return

        for neighbor in graph.neighbors(current_node):
            tentative_g_score = g_score[current_node] + distance(graph, current_node, neighbor)

            if tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                f_score = tentative_g_score + heuristic(neighbor)
                heapq.heappush(priority_queue, (f_score, neighbor))
                visited.add(neighbor)
                parent_map[neighbor] = current_node

    print('\n---\nNo Path Exists\n---\n')
```