

ASSIGNMENT- 9.1

Name : T.Rakshitha

H.T.NO: 2303A51172

Batch : 18

Problem 1:

Consider the following Python function:

```
def find_max(numbers):  
    return max(numbers)
```

Task:

- Write documentation for the function in all three formats:
 - (a) Docstring
 - (b) Inline comments
 - (c) Google-style documentation
- Critically compare the three approaches. Discuss the advantages, disadvantages, and suitable use cases of each style.
- Recommend which documentation style is most effective for a mathematical utilities library and justify your answer

Promt :

```
# Write documentation for the function in three formats
```

```
# and compare them. Also recommend the best style.
```

```
# (a) Docstring Style
```

```
# (b) Inline Comment Style
```

```
# This function takes a list of numbers
```

```
# It uses Python's built-in max() function
```

```
# to return the largest number
```

```
# (c) Google-Style Documentation
```

INPUT:

The screenshot shows a code editor interface with a dark theme. At the top, there are tabs for "Run", "Terminal", and "Help". Below the tabs, there is a search bar with the placeholder "AI" and a "Launch" button. The main area contains three examples of Python code, each demonstrating a different style of documentation:

- (a) Docstring Style:**

```
def find_max_docstring(numbers):
    """
    Returns the maximum value from a list of numbers.

    Parameters:
        numbers (list): A list of numeric values.

    Returns:
        int or float: The largest value in the list.

    Raises:
        ValueError: If the list is empty.
    """
    return max(numbers)
```
- (b) Inline Comment Style:**

```
def find_max_inline(numbers):
    # This function takes a list of numbers

    # It uses Python's built-in max() function
    # to return the largest number

    return max(numbers)
```
- (c) Google-Style Documentation:**

```
def find_max_google(numbers):
    """Returns the maximum value from a list of numbers.

    Args:
        numbers (list[int | float]): A list of numeric values.
```

At the bottom of the editor, there are tabs for "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", "TERMINAL", "PORTS", and "POSTMAN CONSOLE".

The screenshot shows a code editor interface with a dark theme. The top bar includes 'Terminal' and 'Help' buttons, a search bar with 'AI' placeholder, and a tab bar with several tabs: 'ai 9.3(task-2).py', 'ai 9.3(task-3).py', 'ai 9.1(task-1).py' (which is the active tab), 'Launch', 'ai (6.3 task - 5).py', and 'ai(7.1) task'. Below the tabs is a code editor window containing Python code. The code compares three documentation styles: Docstring, Inline Comment, and Google-Style. It includes examples of each style and their pros and cons. The code is color-coded for readability.

```
ai 9.1(task-1).py
ai 9.1(task-1).py > ...
36 def find_max_google(numbers):
44     Raises:
45     |     ValueError: If the list is empty.
46     """
47     return max(numbers)
49
50
51 # ----- Example Execution -----
52
53 numbers = [10, 25, 5, 40, 15]
54
55 print("Docstring Version Output:", find_max_docstring(numbers))
56 print("Inline Comment Version Output:", find_max_inline(numbers))
57 print("Google-Style Version Output:", find_max_google(numbers))
58
59
60 # ----- Comparison & Recommendation Output -----
61
62 print("\nComparison of Documentation Styles:\n")
63
64 print("Docstring:")
65 print("- Structured and accessible using help()")
66 print("- Suitable for reusable libraries")
67 print("- Can be lengthy in large functions\n")
68
69 print("Inline Comments:")
70 print("- Simple and easy to understand")
71 print("- Good for explaining internal logic")
72 print("- Not accessible via help()\n")
73
74 print("Google-Style Documentation:")
75 print("- Professional and highly structured")
76 print("- Clear separation of Args, Returns, and Raises")
77 print("- Suitable for large projects and APIs\n")
78
79 print("Recommended Style: Google-style documentation")
80 print("Reason:")
81 print("- Clearly defines parameters and return values")
82 print("- Improves readability and maintainability")
83 print("- Best suited for mathematical utilities libraries")
```

OUTPUT:

The screenshot shows a terminal window in VS Code with the following text output:

```
PS C:\Users\...\\Desktop\AI> c:\\cd 'c:\\Users\\...\\Desktop\\AI'; & 'C:\\Python314\\python.exe' 'c:\\Users\\...\\vscode\\extensions\\ms-python\\n.debugpy-2025.18.0-win32-x64\\bundled\\libs\\debugpy\\launcher' '51350' '--' 'c:\\Users\\...\\Desktop\\AI\\ai 9.1\\task-1.py'
Docstring Version Output: 40
Inline Comment Version Output: 40
Google-Style Version Output: 40

Comparison of Documentation Styles:

Docstring:
- Structured and accessible using help()
- Suitable for reusable libraries
- Can be lengthy in large functions

Inline Comments:
- Simple and easy to understand
- Good for explaining internal logic
- Not accessible via help()

Google-Style Documentation:
- Professional and highly structured
- Clear separation of Args, Returns, and Raises
- Suitable for large projects and APIs

Recommended Style: Google-style documentation
Reason:
- Clearly defines parameters and return values
- Improves readability and maintainability
- Best suited for mathematical utilities libraries
○ PS C:\\Users\\...\\Desktop\\AI> 
```

EXPLANATION:

Docstring Style – Explanation

Docstrings are written inside triple quotes below a function and provide a clear description of what the function does. They explain inputs, outputs, and possible errors in a readable format. Docstrings can be accessed using `help()` and are useful for documentation tools. They are good for reusable functions and structured code.

Inline Comment Style – Explanation

Inline comments are written using `#` inside the function body. They explain the logic of specific lines of code and improve readability. However, they are not structured documentation and cannot be accessed automatically. They are mainly useful for explaining complex steps in the code.

Google-Style Documentation – Explanation

Google-style documentation is a well-structured docstring format that clearly separates sections like `Args`, `Returns`, and `Raises`. It improves clarity and consistency in larger projects. This style is widely used in professional development. It is best suited for libraries and production-level code.

Problem 2:

Consider the following Python function:

```
def login(user, password, credentials):  
    return credentials.get(user) == password
```

Task:

1. Write documentation in all three formats.
2. Critically compare the approaches.
3. Recommend which style would be most helpful for new developers onboarding a project, and justify your choice.

PROMT:

```
# Write documentation for the login function using standard Python docstring format.  
  
#Rewrite the same login function documentation using inline comments.  
  
#Rewrite the login function using Google-style documentation format.  
  
#Create example credentials data and test all three versions of the function.  
  
#Compare docstring, inline comments, and Google-style documentation  
# in terms of structure, clarity, and maintainability.  
  
# Discuss advantages and disadvantages of each documentation style.  
  
#Recommend the most suitable documentation style  
# for onboarding new developers to a project.  
  
# Justify the recommendation with clear reasoning.
```

INPUT:

The screenshot shows a code editor interface with a dark theme. The top bar includes 'Run', 'Terminal', 'Help', and navigation icons. A search bar contains 'AI'. The main area displays a Python script named 'ents.py'.

```
ents.py
...
ai 9.1(task-2).py > ...
1 # Write documentation for this login function in different formats
2 # and compare them. Finally, suggest which style is best for
3 # new developers joining a project.
4
5
6 # Docstring style documentation
7 def login_docstring(user, password, credentials):
8     """
9     This function checks whether the given username and password
10    match the stored credentials.
11
12    Parameters:
13        user (str): Username of the person logging in.
14        password (str): Password entered by the user.
15        credentials (dict): Dictionary containing username-password pairs.
16
17    Returns:
18        bool: True if login is successful, otherwise False.
19    """
20    return credentials.get(user) == password
21
22
23 # Same function explained using inline comments
24 def login_inline(user, password, credentials):
25     # Take username and password as input
26     # Look up the stored password from the credentials dictionary
27     # Compare stored password with entered password
28     return credentials.get(user) == password
29
30
31 # Google-style documentation format
32 def login_google(user, password, credentials):
33     """
34     Validates user login using stored credentials.
35
36     Args:
37         user (str): Username provided for login.
38         password (str): Password entered by the user.
39         credentials (dict[str, str]): Dictionary storing valid credentials.
40
41     Returns:
42         bool: True if authentication is successful, False otherwise.
43
making it easier for new developers to understand the function quickly.
PS C:\Users\DELL\Desktop\AI>
```

The code illustrates three styles of Python documentation:

- Docstring style documentation:** Used in lines 7-20. It includes a multi-line docstring at the top of the function, parameter descriptions, and a return value description.
- Inline comments:** Used in lines 24-28. It uses triple quotes to explain the function's purpose, its parameters, and its return value.
- Google-style documentation format:** Used in lines 32-43. It uses triple quotes to define arguments and returns.

The status bar at the bottom shows the path 'C:\Users\DELL\Desktop\AI' and a terminal prompt 'PS C:\Users\DELL\Desktop\AI>'.

The screenshot shows a terminal window within a code editor interface. The terminal tab is active at the bottom, displaying the following Python code:

```
Terminal Help ← → Q AI
ai(6.3 task -2).py ai(6.3 task-3).py ai 9.3(task-1) || ⌂ ⌄ ⌅ ⌆ ⌇ ⌈ ⌉ ⌋ ⌊ ⌊ Launch ↴ -3).py ai 9.1(task-1).py ai 9.1(task-2).py > ...
y
32 def login_google(user, password, credentials):
33     return credentials.get(user) == password
34
35
36 # Example data to test the functions
37 credentials_data = {
38     "admin": "1234",
39     "user1": "pass",
40     "john": "abcd"
41 }
42
43 print("Docstring Version:", login_docstring("admin", "1234", credentials_data))
44 print("Inline Version:", login_inline("user1", "wrong", credentials_data))
45 print("Google Version:", login_google("john", "abcd", credentials_data))
46
47 # Comparing the three documentation approaches
48
49 print("\nComparison of Documentation Styles:\n")
50
51 print("Docstring:")
52 print("Structured and visible using help().")
53 print("Good for reusable functions but may become lengthy.\n")
54
55 print("Inline Comments:")
56 print("Easy to understand and helpful for simple logic.")
57 print("Not structured and cannot be accessed using help().\n")
58
59 print("Google-Style:")
60 print("Professional and clearly separates inputs and outputs.")
61 print("Best suited for team projects and larger applications.\n")
62
63 # Recommendation for onboarding new developers
64
65 print("Recommended Style: Google-style documentation.")
66 print("Reason: It clearly explains parameters and return values,")
67 print("making it easier for new developers to understand the function quickly.")
```

The terminal output at the bottom shows the execution of the code, resulting in the message:

```
making it easier for new developers to understand the function quickly.
PS C:\Users\DELL\Desktop\AI>
```

OUTPUT:

```
- Best suited for mathematical utilities libraries
PS C:\Users\DELL\Desktop\AI> c:; cd 'c:\Users\DELL\Desktop\AI'; & 'C:\Python314\python.exe' 'c:\Users\DELL\.vscode\extensions\ms-python
n.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '56666' '--' 'c:\Users\DELL\Desktop\AI\ai 9.1(task-2).py'
Docstring Version: True
Inline Version: False
Google Version: True

Comparison of Documentation Styles:

Docstring:
Structured and visible using help().
Good for reusable functions but may become lengthy.

Inline Comments:
Good for reusable functions but may become lengthy.

Inline Comments:
Inline Comments:
Easy to understand and helpful for simple logic.
Not structured and cannot be accessed using help().

Google-Style:
Professional and clearly separates inputs and outputs.
Best suited for team projects and larger applications.

Recommended Style: Google-style documentation.
Reason: It clearly explains parameters and return values,
making it easier for new developers to understand the function quickly.
PS C:\Users\DELL\Desktop\AI> 
```

EXPLANATION:

Docstrings are written inside triple quotes below a function. They explain the purpose, parameters, and return values clearly. They can be accessed using `help()` and are useful for documentation tools. This style is good for reusable and well-structured code.

Inline Comment Style – Explanation

Inline comments use `#` to explain specific lines of code. They help understand the internal logic step-by-step. However, they are not structured and cannot be accessed as formal documentation. They are best for explaining simple or complex logic inside the function.

Google-Style Documentation – Explanation

Google-style documentation is a structured docstring format with clear sections like `Args` and `Returns`. It improves clarity and consistency in team projects. It is widely used in industry and suitable for larger applications.

Recommendation

Google-style documentation is best for new developers because it clearly explains inputs and outputs in a structured way, making the function easy to understand quickly

Problem 3: Calculator (Automatic Documentation Generation)

Task: Design a Python module named calculator.py and demonstrate automatic documentation generation.

Instructions:

1. Create a Python module calculator.py that includes the following functions, each written with appropriate docstrings:
 - o add(a, b) – returns the sum of two numbers
 - o subtract(a, b) – returns the difference of two numbers
 - o multiply(a, b) – returns the product of two numbers
 - o divide(a, b) – returns the quotient of two numbers
2. Display the module documentation in the terminal using Python's documentation tools.
3. Generate and export the module documentation in HTML format using the pydoc utility, and open the generated HTML file in a web browser to verify the output.

PROMT:

```
# Prompt: Create basic arithmetic functions (add, subtract, multiply, divide).  
# Prompt: Add simple docstrings to each function for automatic documentation generation.  
# Prompt: Use help() to display documentation of each function in the terminal.  
# Prompt: Execute each function with sample input values.  
# Prompt: Display the output of all arithmetic operations clearly.
```

INPUT :

The screenshot shows a Jupyter Notebook interface with the following details:

- Toolbar:** Run, Terminal, Help, back/forward buttons, AI search bar.
- File List:** ai(6.3 task-3).py, ai 9.3(task-1).py, ai 9.3(task-2), ai 9.1(task-2).py, ai 9.1(task-3).py.
- Code Cell:** Contains Python code for a calculator with four operations: add, subtract, multiply, and divide. It includes docstrings and demonstrates function execution with sample values.
- Output Cell:** Shows the results of the division operation: Divide: 2.0
- Bottom Navigation:** PROBLEMS, OUTPUT, DEBUG CONSOLE, TERMINAL (selected), PORTS, POSTMAN CONSOLE.

```
... ai(6.3 task-3).py ai 9.3(task-1).py ai 9.3(task-2) || ⌂ ⌃ ⌄ ⌅ ⌆ Launch <-1>.py ai 9.1(task-2).py ai 9.1(task-3).py

ai 9.1(task-3).py > ...
1 # Prompt: Create basic arithmetic functions with proper docstrings.
2
3 def add(a, b):
4     """Return the sum of two numbers."""
5     return a + b
6
7
8 def subtract(a, b):
9     """Return the difference of two numbers."""
10    return a - b
11
12
13 def multiply(a, b):
14     """Return the product of two numbers."""
15     return a * b
16
17
18 def divide(a, b):
19     """Return the quotient of two numbers."""
20     if b == 0:
21         return "Cannot divide by zero"
22     return a / b
23
24
25 # Prompt: Demonstrate automatic documentation generation using help().
26
27 print("==> Calculator Documentation ==\n")
28 help(add)
29 help(subtract)
30 help(multiply)
31 help(divide)
32
33
34 # Prompt: Demonstrate function execution with sample values.
35
36 print("\n==> Function Outputs ==")
37 print("Add:", add(10, 5))
38 print("Subtract:", subtract(10, 5))
39 print("Multiply:", multiply(10, 5))
40 print("Divide:", divide(10, 5))

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTMAN CONSOLE

Divide: 2.0
PS C:\Users\DoLL\Desktop\AT>
```

OUTPUT:

```
55
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    PORTS    POSTMAN CONSOLE

○
Help on function divide in module __main__:

divide(a, b)
    Return the quotient of two numbers.
divide(a, b)
    Return the quotient of two numbers.

Return the quotient of two numbers.

--- Function Outputs ---
--- Function Outputs ---
Add: 15
Subtract: 5
Multiply: 50
Divide: 2.0
PS C:\Users\DELL\Desktop\AI> 
```

EXPLANATION:

Docstring Explanation

In this program, each arithmetic function contains a simple docstring that briefly explains its purpose. The docstring is written inside triple quotes directly below the function definition. When the help() function is called, Python automatically displays this documentation. This demonstrates how docstrings support automatic documentation generation.

Importance of Using Docstrings

Docstrings make functions self-explanatory and improve readability. They allow developers to understand what a function does without reading its implementation. Using help() is especially useful in larger projects, libraries, and team environments. This approach ensures clean, professional, and maintainable code.

Problem 4: Conversion Utilities Module

Task:

1. Write a module named conversion.py with functions:
 - o decimal_to_binary(n)
 - o binary_to_decimal(b)
 - o decimal_to_hexadecimal(n)
2. Use Copilot for auto-generating docstrings.
3. Generate documentation in the terminal.
4. Export the documentation in HTML format and open it in a

Browser

PROMPTS:

```
# Create a Python module named conversion.py with number conversion functions.

# Implement decimal_to_binary(n) to convert a decimal number to binary.

# Implement binary_to_decimal(b) to convert a binary string to decimal.

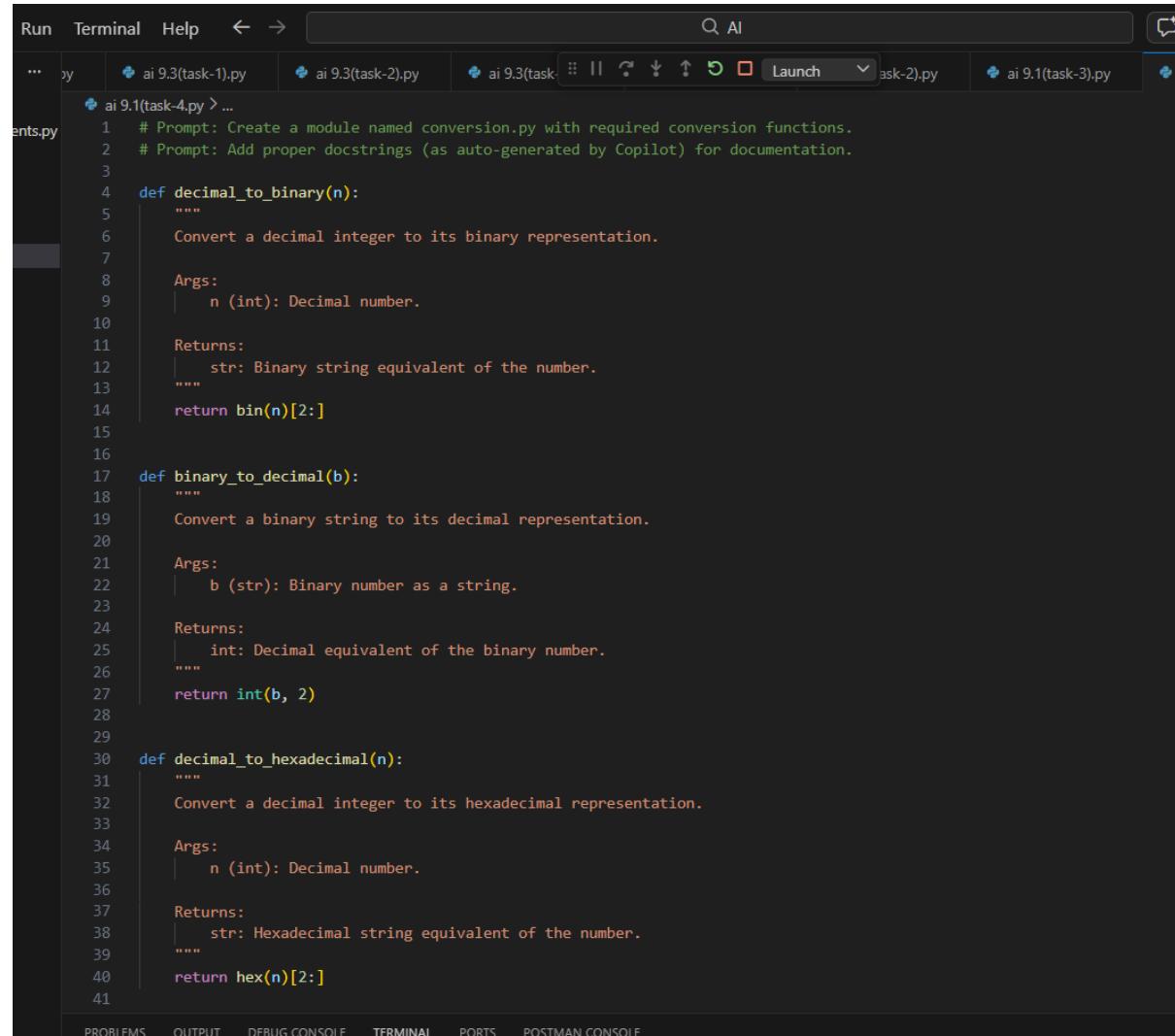
# Implement decimal_to_hexadecimal(n) to convert a decimal number to hexadecimal.

# Add proper docstrings to all functions (as if generated by Copilot).

# Use help() to display function documentation in the terminal.
```

```
# Demonstrate the conversion functions using sample inputs.  
# Generate module documentation in HTML format using the pydoc utility.  
# Open the generated HTML file in a web browser to verify the documentation.
```

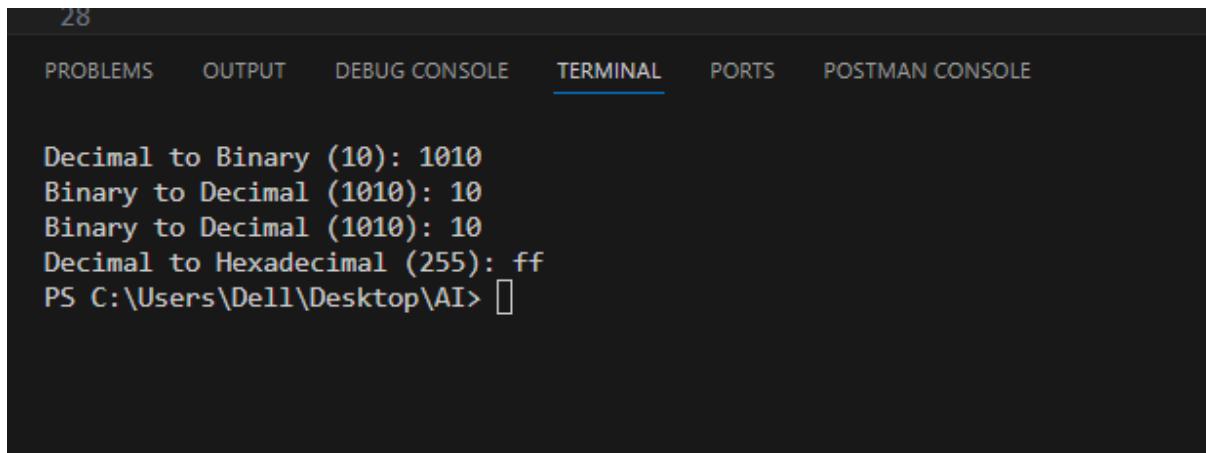
INPUT:



The screenshot shows a code editor interface with a dark theme. The top menu bar includes 'Run', 'Terminal', 'Help', and various navigation icons. A search bar labeled 'AI' is positioned at the top right. Below the menu, there are tabs for multiple files: '... py', 'ai 9.3(task-1).py', 'ai 9.3(task-2).py', 'ai 9.3(task-4.py) > ...', 'ai 9.1(task-2).py', and 'ai 9.1(task-3).py'. The current file is 'ai 9.1(task-4.py)'. The code itself is a Python module named 'conversion.py' containing three functions: `decimal_to_binary`, `binary_to_decimal`, and `decimal_to_hexadecimal`. Each function includes a docstring with detailed descriptions of its parameters ('Args') and return values ('Returns'). The code uses triple quotes for the docstrings and standard Python syntax for function definitions and returns.

```
... py | ai 9.3(task-1).py | ai 9.3(task-2).py | ai 9.3(task-4.py) > ... | ai 9.1(task-2).py | ai 9.1(task-3).py  
ents.py  
1 # Prompt: Create a module named conversion.py with required conversion functions.  
2 # Prompt: Add proper docstrings (as auto-generated by Copilot) for documentation.  
3  
4 def decimal_to_binary(n):  
    """  
        Convert a decimal integer to its binary representation.  
    """  
    Args:  
        n (int): Decimal number.  
    Returns:  
        str: Binary string equivalent of the number.  
    """  
    return bin(n)[2:]  
15  
16  
17 def binary_to_decimal(b):  
    """  
        Convert a binary string to its decimal representation.  
    """  
    Args:  
        b (str): Binary number as a string.  
    Returns:  
        int: Decimal equivalent of the binary number.  
    """  
    return int(b, 2)  
28  
29  
30 def decimal_to_hexadecimal(n):  
    """  
        Convert a decimal integer to its hexadecimal representation.  
    """  
    Args:  
        n (int): Decimal number.  
    Returns:  
        str: Hexadecimal string equivalent of the number.  
    """  
    return hex(n)[2:]  
41
```

OUTPUT:



The screenshot shows a terminal window with the following text output:

```
Decimal to Binary (10): 1010
Binary to Decimal (1010): 10
Binary to Decimal (1010): 10
Decimal to Hexadecimal (255): ff
PS C:\Users\DELL\Desktop\AI> 
```

Explanation:

Each function in the module contains a structured docstring that explains its purpose, input arguments, and return values. These docstrings follow a clear format similar to professional documentation standards. When help() is used, Python automatically reads and displays these docstrings in the terminal. This makes the module self-documented and easy to understand.

Automatic Documentation Generation

The help() function demonstrates how Python can generate documentation directly from docstrings. Additionally, using the pydoc -w conversion command creates an HTML file containing the module documentation. This shows how properly written docstrings support both terminal-based and web-based documentation generation, making the code more professional and maintainable.

Problem 5 – Course Management Module

Task:

1. Create a module course.py with functions:

- o add_course(course_id, name, credits)
- o remove_course(course_id)
- o get_course(course_id)

2. Add docstrings with Copilot.

3. Generate documentation in the terminal.

4. Export the documentation in HTML format and open it in a browser.

PROMPTS:

COURSE MANAGEMENT MODULE

```
# This module should allows adding, removing, and retrieving  
# course details. It also demonstrates documentation  
# generation using help().
```

```
# In-memory course storage
```

INPUT:

The screenshot shows a code editor interface with a dark theme. The top bar has tabs for 'c-2.py', 'ai 9.3(task-3).py', 'ai 9.1(task-1).py', 'ai 9.1', and '9.1(task-4.py)'. Below the tabs is a toolbar with icons for file operations like 'New', 'Open', 'Save', 'Print', and 'Launch'. The main area contains the following Python code:

```
1 # COURSE MANAGEMENT MODULE  
2 # This module should allows adding, removing, and retrieving  
3 # course details. It also demonstrates documentation  
4 # generation using help().  
5  
6  
7 # In-memory course storage  
8 courses = []  
9  
10 def add_course(course_id, name, credits):  
11     """  
12         Add a new course to the course management system.  
13     Args:  
14         course_id (str): Unique identifier for the course.  
15         name (str): Name of the course.  
16         credits (int): Credit value assigned to the course.  
17     Returns:  
18         str: Success or error message based on operation result.  
19     """  
20     if course_id in courses:  
21         return "Course ID already exists."  
22  
23     if credits <= 0:  
24         return "Credits must be a positive number."  
25  
26     courses[course_id] = {  
27         "name": name,  
28         "credits": credits  
29     }  
30     return "Course added successfully."  
31  
32 def remove_course(course_id):  
33     """  
34         Remove an existing course from the system.  
35     Args:  
36         course_id (str): Unique identifier of the course.
```

At the bottom of the editor, there are tabs for 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is underlined), 'PORTS', and 'POSTMAN CONSOLE'.

The screenshot shows a code editor interface with a dark theme. At the top, there is a navigation bar with 'Run', 'Terminal', and 'Help' options, followed by file tabs: 'k-2.py', 'ai 9.3(task-3).py', 'ai 9.1(task-1).py', 'ai 9.1 (task-5).py > ...', 'ai 9.1 (task-4.py)', and a search bar labeled 'AI'. Below the tabs is a toolbar with icons for file operations like copy, paste, and save.

The main area displays a Python script named 'courses.py'. The code defines two functions: `remove_course` and `get_course`. It also contains a main block that prints a welcome message and a menu loop. The code is annotated with docstrings and comments explaining its purpose and arguments.

```
36     def remove_course(course_id):
37         """
38             Returns:
39                 str: Success or error message.
40             """
41         if course_id in courses:
42             del courses[course_id]
43             return "Course removed successfully."
44         return "Course not found."
45
46     def get_course(course_id):
47         """
48             Retrieve course details using course ID.
49
50             Args:
51                 course_id (str): Unique identifier of the course.
52
53             Returns:
54                 dict or str: Course details if found, otherwise error message.
55             """
56
57         return courses.get(course_id, "Course not found.")
58
59
60
61
62
63
64
65
66     if __name__ == "__main__":
67
68         print("====")
69         print("      COURSE MANAGEMENT SYSTEM")
70         print("====")
71
72         while True:
73
74             print("\nPlease choose an option below:")
75             print("1. Add a New Course")
76             print("2. Remove an Existing Course")
77             print("3. View Course Details")
78             print("4. Show Function Documentation")
79             print("5. Exit")
80
81             choice = input("Enter your choice (1-5): ")
```

At the bottom of the editor, there is a navigation bar with tabs: 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', 'TERMINAL' (which is currently selected), 'PORTS', and 'POSTMAN CONSOLE'. There is also a '... Exit' button.

The screenshot shows a code editor interface with a dark theme. At the top, there's a navigation bar with tabs for "Terminal", "Help", and several file icons. To the right of the tabs is a search bar with the placeholder "Q AI". Below the navigation bar is a toolbar with icons for file operations like copy, paste, and save.

The main area contains the code for a Python script named `k-2.py`. The code is a script for a Course Management System. It includes functions for adding, removing, and getting course details, as well as documentation generation and exiting the system. The code uses standard Python syntax with imports and functions like `input()`, `print()`, and `break`.

```
109     # GET COURSE DETAILS
110
111     elif choice == "3":
112
113         course_id = input("\nEnter Course ID to view details: ").strip()
114         result = get_course(course_id)
115
116         if isinstance(result, dict):
117             print("\nCourse Details:")
118             print("Course ID : ", course_id)
119             print("Name      : ", result["name"])
120             print("Credits   : ", result["credits"])
121         else:
122             print(result)
123
124
125     # SHOW DOCUMENTATION USING help()
126
127     elif choice == "4":
128
129         print("\n==== FUNCTION DOCUMENTATION ====\n")
130         help(add_course)
131         help(remove_course)
132         help(get_course)
133
134         print("\nTo export documentation in HTML format, run this in terminal:")
135         print("pydoc -w course")
136
137         print("\nThen open the generated file using:")
138         print("Windows : start course.html")
139         print("Mac      : open course.html")
140
141
142     elif choice == "5":
143         print("\nExiting Course Management System...")
144         break
145
146
147     else:
148         print("Invalid choice. Please select between 1 and 5.")
```

At the bottom of the editor, there are tabs for "PROBLEMS", "OUTPUT", "DEBUG CONSOLE", "TERMINAL", "PORTS", and "POSTMAN CONSOLE".

OUTPUT:

```
Run Terminal Help ← → ⌂ AI
... PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS POSTS Launch + ⌂ power
ments.py PS C:\Users\... & 'C:\Python314\python.exe' 'c:\Users\...extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '52065' '--' 'c:\Users\...Desktop\AI\try.py'
PS C:\Users\... & cd 'c:\Users\...Desktop\AI'; & 'C:\Python314\python.exe' 'c:\Users\...extensions\ms-python.debugpy-2025.18.0-win32-x64\bundled\libs\debugpy\launcher' '56235' '--' 'c:\Users\...Desktop\AI\ai 9.1 (task-5).py'
===== COURSE MANAGEMENT SYSTEM =====
Please choose an option below:
1. Add a New Course
2. Remove an Existing Course
3. View Course Details
4. Show Function Documentation
5. Exit
Enter your choice (1-5): 1
Enter Course ID: 2345678
Enter Course Name: ai
Enter Course Credits: 3
Enter your choice (1-5): 1
Enter Course ID: 2345678
Enter Course Name: ai
Enter Course Credits: 3
Enter Course ID: 2345678
Enter Course Name: ai
Enter Course Credits: 3
Enter Course Name: ai
Enter Course Credits: 3
Enter Course Credits: 3
Course added successfully.

Please choose an option below:
1. Add a New Course
2. Remove an Existing Course
3. View Course Details
4. Show Function Documentation
5. Exit
Enter your choice (1-5):
```

Explanation:

This program is a simple course management system that allows users to add, remove, and view course details through a menu-driven interface. It stores course information in a dictionary and performs basic validation like checking duplicate IDs and valid credits. Each function includes proper docstrings to support documentation generation. The documentation can be viewed using `help()` or exported to HTML format using the `pydoc` command.