Source code:

```
graph = {'A':['B','C'],'B':['D','E'],'C':['F'],'D':[],'E':[],'F':[]}
visited = []
queue = []
def bfs(start):
    visited.append(start)
    queue.append(start)
    while queue:
        node = queue.pop(0)
        print(node, end=" ")
        for n in graph[node]:
            if n not in visited:
                visited.append(n)
                queue.append(n)
bfs('A')
```

OUTPUT:

```
A B C D E F
|
```

SOURCE CODE:

```
graph = {'A':['B','C'],'B':['D','E'],'C':['F'],'D':[],'E':[],'F':[]}
visited = set()
def dfs(node):
    if node not in visited:
        print(node, end=" ")
        visited.add(node)
        for n in graph[node]:
            dfs(n)
dfs('A')
```

OUTPUT:

```
A B D E C F
```

SOURCE CODE:

```
graph = {'A':[('B',1),('C',3)],'B':[('D',1)],'C':[('D',1)],'D':[]}
h = {'A':3,'B':1,'C':1,'D':0}
open_list = ['A']
g = {'A':0}
while open_list:
    node = min(open_list, key=lambda x: g[x] + h[x])
    print(node, end=" ")
    if node == 'D':
        break
    open_list.remove(node)
    for (n,c) in graph[node]:
      g[n] = g[node] + c
      open_list.append(n)
```

OUTPUT:

```
A B D
|
```

SOURCE CODE:

```
def minimax(depth, nodeIndex, isMax, values, height):
    if depth == height:
        return values[nodeIndex]
    if isMax:
        return max(minimax(depth+1,nodeIndex*2,False,values,height),
minimax(depth+1,nodeIndex*2+1,False,values,height))
    else:
        return min(minimax(depth+1,nodeIndex*2,True,values,height),
minimax(depth+1,nodeIndex*2+1,True,values,height))
values = [3,5,6,9,1,2,0,-1]
print(minimax(0,0,True,values,3))
```

OUTPUT:

SOURCE CODE:

```
P_D = 0.01

P_Pos_D = 0.99

P_Pos_notD = 0.05

P_notD = 1 - P_D

P_Pos = (P_Pos_D * P_D) + (P_Pos_notD * P_notD)

P_D_Pos = (P_Pos_D * P_D) / P_Pos

print(P_D_Pos)
```

OUTPUT:

```
0.16666666666666669
```

SOURCECODE:

```
colors = ["Red","Green","Blue"]
graph = {'A':['B','C'],'B':['A','C'],'C':['A','B']}
solution = {}
def safe(node, color):
    for n in graph[node]:
        if n in solution and solution[n] == color:
            return False
    return True
 def solve(node):
     if node == len(graph):
            return True
     nodes = list(graph.keys())
     for color in colors:
         if  safe(nodes[node], color):
             solution[nodes[node]] = color
             if solve(node+1):
                    return True
             del solution[nodes[node]]
             return False
solve(0)
print(solution)
```

OUTPUT:

```
{'A': 'Red', 'B': 'Green', 'C': 'Blue'}
```

SOURCE CODE:

```
states = [0,1]
rewards = {0:5,1:10}
gamma = 0.9
V = [0,0]
for i in range(5):
    newV = V.copy()
    for s in states:
        newV[s] = rewards[s] + gamma * V[s]
    V = newV
print(V)
```

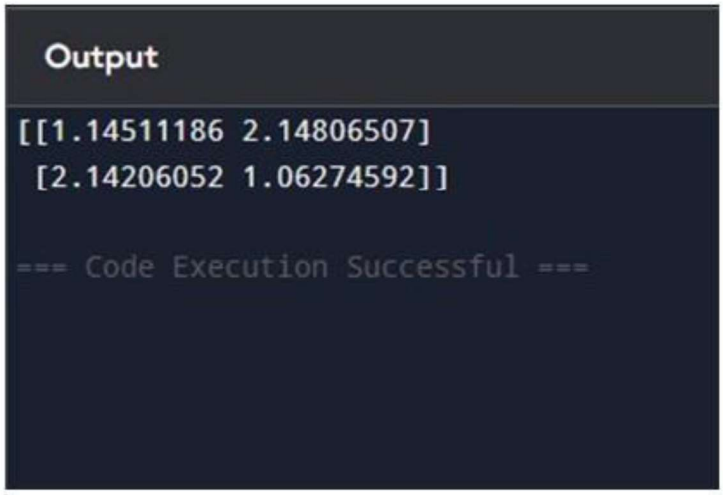
OUTPUT:

```
[20.4755, 40.951]
```

SOURCE CODE:

```
import numpy as np

Q = np.zeros((2,2))

gamma = 0.8

alpha = 0.1

rewards = [[0,1],[1,0]]

for i in range(100):

    s = np.random.randint(0,2)

    a = np.random.randint(0,2)

    Q[s,a] = Q[s,a] + alpha * ( rewards[s][a] + gamma * np.max(Q[a]) - Q[s,a] )

print(Q)
```

OUTPUT:



1. # Simple representation of Software Process

```
def requirements():
    return "Requirements gathered successfully."

def design():
    return "System design completed."

def implementation():
    return "Coding phase completed."

def testing():
    return "Testing done successfully."

def deployment():
    return "Software deployed."

def maintenance():
    return "Maintenance activity ongoing."

# Main process
print("Software Process Execution:\n")

print(requirements())
print(design())
print(implementation())
print(testing())
print(deployment())
print(maintenance())
```

Output

```
Software Process Execution:

Requirements gathered successfully.
System design completed.
Coding phase completed.
Testing done successfully.
Software deployed.
```

2. # Software Process Model Selector

```
def select_model(requirements_clear, project_size, need_flexibility):
    if requirements_clear and project_size == "large":
        return "Waterfall Model"
    elif not requirements_clear and need_flexibility:
        return "Agile Model"
    elif project_size == "medium":
        return "Incremental Model"
    else:
        return "Spiral Model"

# Example inputs
requirements_clear = False
project_size = "small"
need_flexibility = True

model = select_model(requirements_clear, project_size, need_flexibility)

print("Recommended Software Process Model:", model)
```

Output

```
Recommended Software Process Model: Agile Model
```

3.# Agile vs Conventional Software Development Simulation

```
def waterfall_process():
    print("WATERFALL MODEL")
    stages = ["Requirement Analysis", "Design", "Implementation", "Testing", "Deployment"]

    for stage in stages:
        print(f"Completed: {stage}")

    print("Final Product Delivered\n")


def agile_process(sprints):
    print("AGILE MODEL")

    for i in range(1, sprints + 1):
        print(f"--- Sprint {i} ---")
        print("Planning")
        print("Design")
        print("Development")
        print("Testing")
        print("Deliver Working Increment\n")


# Run both models
waterfall_process()
agile_process(3)
```

Output

```
WATERFALL MODEL
Completed: Requirement Analysis
Completed: Design
Completed: Implementation
Completed: Testing
Completed: Deployment
Final Product Delivered
```

4# Software Engineering Practices Framework Simulation

```
def communication():
    print("1. COMMUNICATION PHASE")
    print("Requirements gathered from client.")
    print("SRS document prepared.\n")


def planning():
    print("2. PLANNING PHASE")
    print("Project schedule created.")
    print("Resources and cost estimated.\n")


def modeling():
    print("3. MODELING PHASE")
    print("System design created using UML diagrams.\n")


def construction():
    print("4. CONSTRUCTION PHASE")
    print("Coding completed.")
    print("Unit and integration testing done.\n")


def deployment():
    print("5. DEPLOYMENT PHASE")
    print("Software installed for users.")
    print("Maintenance and support started.\n")


# Execute all phases
communication()
planning()
modeling()
construction()
deployment()
```

Output

1. COMMUNICATION PHASE
Requirements gathered from client.
SRS document prepared.

2. PLANNING PHASE
Project schedule created.
Resources and cost estimated.

3. MODELING PHASE
System design created using UML diagrams.

4. CONSTRUCTION PHASE
Coding completed.
Unit and integration testing done.

5. DEPLOYMENT PHASE
Software installed for users.
Maintenance and support started.

5.# Simulation of Communication and Planning in Software Engineering

```
def communication():
    print("COMMUNICATION PHASE")
    print("Client requirements gathered for Online Shopping App.")
    print("Features identified: Login, Cart, Payment, Delivery Tracking.")
    print("SRS document prepared.\n")


def planning():
    print("PLANNING PHASE")
    print("Project duration estimated: 3 months")
    print("Team assigned: 4 developers, 1 tester")
    print("Cost estimated and resources allocated.")
    print("Risk identified: Payment gateway failure\n")


# Execute phases
communication()
planning()
```

Output

```
COMMUNICATION PHASE
Client requirements gathered for Online Shopping App.
Features identified: Login, Cart, Payment, Delivery Tracking.
SRS document prepared.

PLANNING PHASE
Project duration estimated: 3 months
Team assigned: 4 developers, 1 tester
Cost estimated and resources allocated.
Risk identified: Payment gateway failure
```

6.# System Engineering Hierarchy Simulation

```
def system_engineering():
    print("SYSTEM ENGINEERING HIERARCHY\n")

    print("System: Banking System")

    subsystems = {
        "Account Management": ["Login Module", "Profile Module"],
        "Transaction System": ["Deposit Module", "Withdraw Module"],
        "Security System": ["Authentication Module", "Encryption Module"]
    }

    for subsystem, modules in subsystems.items():
        print(f"\n Subsystem: {subsystem}")
        for module in modules:
            print(f"   -> Module: {module}")
            print(f"      -> Component: Basic processing functions")


system_engineering()
```

Output

```
SYSTEM ENGINEERING HIERARCHY
System: Banking System

 Subsystem: Account Management
   -> Module: Login Module
      -> Component: Basic processing functions
   -> Module: Profile Module
      -> Component: Basic processing functions

 Subsystem: Transaction System
   -> Module: Deposit Module
      -> Component: Basic processing functions
   -> Module: Withdraw Module
      -> Component: Basic processing functions

 Subsystem: Security System
   -> Module: Authentication Module
      -> Component: Basic processing functions
   -> Module: Encryption Module
      -> Component: Basic processing functions
```

7.# Business Process Engineering Simulation

```
class BusinessProcess:
    def __init__(self, name):
        self.name = name
        self.steps = []

    def add_step(self, step):
        self.steps.append(step)

    def execute(self):
        print(f"\nExecuting Business Process: {self.name}")
        for i, step in enumerate(self.steps, start=1):
            print(f"Step {i}: {step}")
        print("Process Completed Successfully!\n")


# Product Engineering Simulation

class ProductEngineering:
    def __init__(self, product_name):
        self.product_name = product_name
        self.stages = ["Requirement Analysis", "Design", "Development", "Testing",
"Deployment"]

    def build_product(self):
        print(f"\nBuilding Product: {self.product_name}")
        for stage in self.stages:
            print(f"Stage: {stage}")
        print("Product Successfully Developed!\n")


# Example usage

# Business Process Engineering Example
order_process = BusinessProcess("Online Order Processing")
order_process.add_step("Receive Order")
order_process.add_step("Check Inventory")
order_process.add_step("Process Payment")
order_process.add_step("Dispatch Product")
order_process.execute()
```

```
# Product Engineering Example
app = ProductEngineering("Mobile Banking App")
app.build_product()
```

Output

```
Executing Business Process: Online Order Processing
Step 1: Receive Order
Step 2: Check Inventory
Step 3: Process Payment
Step 4: Dispatch Product
Process Completed Successfully!


Building Product: Mobile Banking App
Stage: Requirement Analysis
Stage: Design
Stage: Development
Stage: Testing
Stage: Deployment
Product Successfully Developed!
```

8. # WebEngineeringProcess

```
class WebEngineeringProcess:
    def __init__(self, project_name):
        self.project_name = project_name
        self.phases = ["Communication", "Planning", "Modeling", "Construction",
"Deployment"]

    def execute_process(self):
        print(f"\nWeb Engineering Process for: {self.project_name}")
        for phase in self.phases:
            print(f"Phase: {phase}")
        print("Web Application Development Completed Successfully!\n")


class WebAnalysisModels:
    def show_models(self):
        print("Web Application Analysis Models:\n")

        print("1. Content Model: Defines website content like text, images, videos.")
        print("2. Navigation Model: Defines user flow between pages.")
        print("3. Interaction Model: Defines user interactions like forms and clicks.")
        print("4. Architecture Model: Defines system structure (client-server).")
        print("5. Functional Model: Defines system functions like login, search.\n")


# Example execution

project = WebEngineeringProcess("Online Shopping Website")
project.execute_process()

models = WebAnalysisModels()
models.show_models()
```

Output

Web Engineering Process for: Online Shopping Website
Phase: Communication
Phase: Planning
Phase: Modeling
Phase: Construction
Phase: Deployment
Web Application Development Completed Successfully!

Web Application Analysis Models:

1. Content Model: Defines website content like text, images, videos.
2. Navigation Model: Defines user flow between pages.
3. Interaction Model: Defines user interactions like forms and clicks.
4. Architecture Model: Defines system structure (client-server).
5. Functional Model: Defines system functions like login, search.