In [ ]:
```
%matplotlib inline
```

Assignment 4

DUE: Friday, March 10th at 5:00pm

Turn in the assignment via Canvas.

To write legible answers you will need to be familiar with both Markdown and Latex

Before you turn this problem in, make sure everything runs as expected. To do so, restart the kernel and run all cells (in the menubar, select Runtime→→Restart and run all).

Show your work!

Whenever you are asked to find the solution to a problem, be sure to also **show how you arrived** at your answer.

In [ ]:
```
%matplotlib inline
```

In [ ]:
```
NAME = "Sathyaprakash Narayanan"
STUDENT_ID = "2005873"
```

Problem 1 - Bayes' Theorem - Testing for a Genetic Defect

Testing for a genetic defect is not perfect. There are two kinds of errors in a binary test, a **false positive** where a test result incorrectly indicates a presence of a condition when it is not present, and a **false negative** where the test incorrectly fails to indicate the presence of a condition when it is present. These are in contrast with the two correct results; **true positive** and **true negative**.

Given the following data, answer the following questions

- Two percent of people have this genetic defect.
- Ninety two percent of tests for the gene detect it (true positive)
- Six percent of tests are false positives.

Part 1

What is the probability of getting a positive result on the test?

P(defect) = 0.02

P(positive | defect) = 0.92

P(positive) = P(positive | defect) *P(defect) + P(positive | no defect)* P(no defect)

P(positive) = 0.92 *0.02 + 0.06* (1 - 0.02)

P(positive) = 0.0696 + 0.0588

P(positive) = 0.1284

Part 2

What is the probability of having the gene, given a positive result?

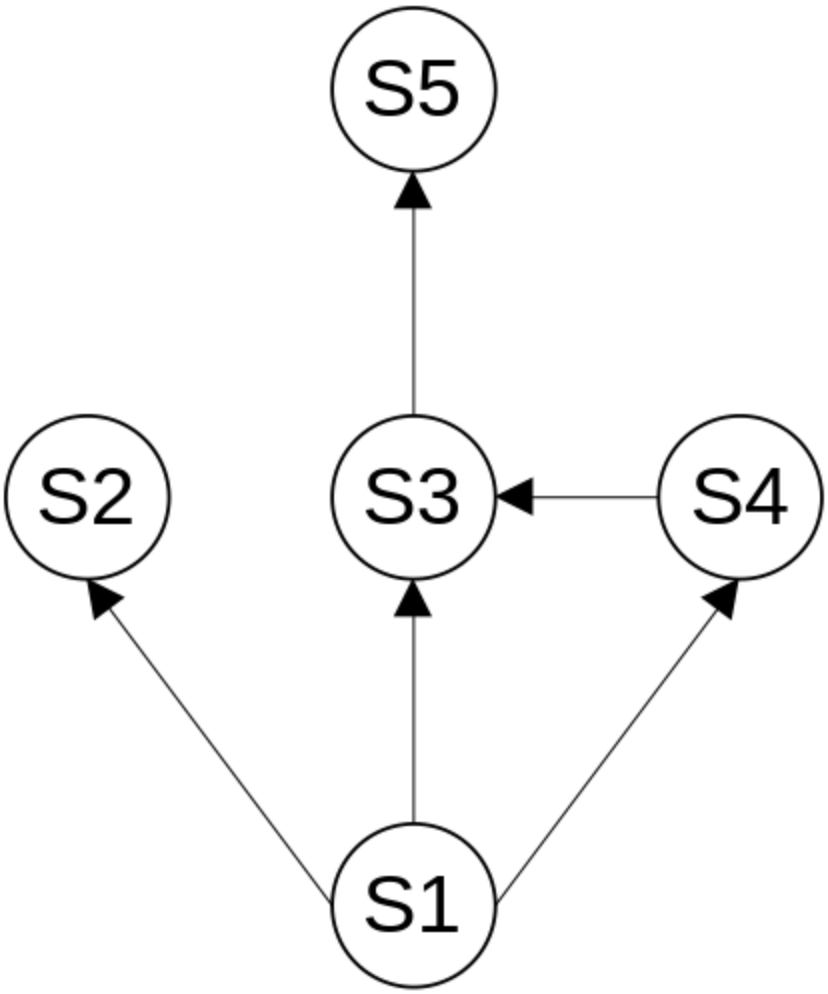
P(defect | positive) = P(positive | defect) * P(defect) / P(positive)

P(defect | positive) = 0.92 * 0.02 / 0.1284

P(defect | positive) = 0.0142

Problem 2 - Bayesian Networks

You are given the following Bayyesian network consisting of five binary random variables S1, S2, S3, S4, S5. Each variable can represent a certain state which can be "True" or "False". Answer the following questions based on the probabilities given below.



- P(S1 = true) = 0.5
- P(S2 = true | S1) = {0.6, S1 = false; 0.75, S1 = true}
- P(S4 = true | S1) = {0.1, S1 = false; 0.8, S1 = true}

- P(S3 = true | S1,S4) = {0.2, S1=false,S4=false; 0.8, S1=true,S4=false; 0.4, S1=false,S4=true; 0.9, S1=true,S4=true}
- P(S5 = true | S3) = { 0.75, S3 = false; 0.1 S3 = true}

Part 1

Given the above network and probabilities, calculate the probability of P(S1=true, S2=true, S3=true, S4=true, S5=true).

P(S1=true, S2=true, S3=true, S4=true, S5=true) = P(S5=true | S3=true) *P(S3=true | S1=true, S4=true)* P(S4=true | S1=true) *P(S2=true | S1=true)* P(S1=true)

==> 0.1 *0.9* 0.8 *0.75* 0.5 = **0.027**

Part 2

Given the above network and probabilities, calculate the probability of P(S5 = true | S1 = true).

P(S5 = true | S3 = false) = 0.75 P(S5 = true | S3 = true) = 0.1

P(S3 = true | S1 = true, S4 = true) = 0.9

P(S3 = true | S1 = true) = P(S3 = true | S1 = true, S4 = true) *P(S4 = true | S1 = true) + P(S3 = true | S1 = true, S4 = false)* P(S4 = false | S1 = true) = 0.9 *0.8 + 0.8* 0.2 = 0.86

P(S5 = true | S1 = true) = 0.75 *(1 - 0.86) + 0.1* 0.86 = 0.088

Part 3

Given the above network and probabilities, calculate the probability of P(S1 = true | S5 = true).

P(S1 = true | S5 = true) = P(S5 = true | S1 = true) * P(S1 = true) / P(S5 = true)

P(S5 = true | S1 = true) = P(S5 = true | S3 = false) *P(S3 = false | S1 = true)* P(S1 = true) + P(S5 = true | S3 = true) *P(S3 = true | S4 = true, S1 = true)* P(S4 = true | S1 = true) *P(S1 = true) + P(S5 = true | S3 = true)* P(S3 = true | S4 = false, S1 = true) *P(S4 = false | S1 = true)* P(S1 = true) = 0.75 *(1 - 0.86)* 0.5 + 0.1 *0.9* 0.8 *0.5 + 0.1* 0.8 *0.5* 0.5 = 0.074

P(S5 = true) = P(S5 = true | S3 = false) *P(S3 = false | S1 = false)* P(S1 = false) + P(S5 = true | S3 = false) *P(S3 = false | S1 = true)* P(S1 = true) + P(S5 = true | S3 = true) *P(S3 = true | S4 = false, S1 = false)* P(S4 = false | S1 = false) *P(S1 = false) + P(S5 = true | S3 = true)* P(S3 = true | S4 = false, S1 = true) *P(S4 = false | S1 = true)* P(S1 = true) + P(S5 = true | S3 = true) *P(S3 = true | S4 = true, S1 = false)* P(S4 = true | S1 = false) *P(S1 = false) + P(S5 = true | S3 = true)* P(S3 = true | S4 = true, S1 = true) *P(S4 = true | S1 = true)* P(S1 = true) = 0.75 *0.8* 0.5 + 0.75 *(1 - 0.86)* 0.5 + 0.1 *0.6* 0.5 *0.5 + 0.1* 0.8 *0.5* 0.5 + 0.1 *0.2* 0.5 *0.5 + 0.1* 0.9 *0.8* 0.5 = 0.3425

P(S1 = true | S5 = true) = 0.074 / 0.3425 = 0.2153

Part 4

Using the chain rule of probability factor P(S1, S2, S3, S4, S5) give the minimum number of parameters to specify the distribution according to the independencies mentioned:

(i) S1, S2, S3, S4, S5 are mutually independent.

(ii) no independencies.

```
i) P(S1, S2, S3, S4, S5) = P(S1) * P(S2) * P(S3) * P(S4) * P(S5)

Since all the variables are mutually independent,
 we do not need any parameters to specify the distribution other than
the probabilities of each individual variable.

ii) P(S1), P(S2 | S1), P(S3 | S1, S2), P(S4 | S1), P(S5 | S1, S2, S3,
S4)

This results in a total of 2^5 - 1 = 31 parameters
```

Problem 3 - COVID-19 Simulation

Using MDP, model the spread of COVID-19 for creating a simulator, under certain assumptions in a hypothetical situation. In the simulation, the inhabitants of a town move around as they go about their regular business.

In the simulation, there are 2 types of people in the town, a **carrier** or **non-carrier** of the Coronavirus. The virus spreads via droplets when an infected person comes in close contact (within 6 feets) of another person who has the virus, a carrier. Non-carriers cannot transmit the disease. Associated with these two types are 5 different states that a person can be in. A carrier can be in one of the following 2 states: infected (but not sick i.e. asymptomatic), or sick. A non-carrier can be in 3 states: unexposed, dead, or immune.

The states (with a distinct color representing the state in the MDP) are described as the following:

- **Unexposed** (BLUE): The people who haven't encountered the virus at all so far.
- **Infected** (ORANGE): People who have the virus but have no symptoms of the disease. An unexposed person has an 80% chance of getting infected on coming in contact with a person carrying the virus. An infected person may get sick with symptoms in 5 days or stay infected (contagious) for 15 days and develop immunity after that.
- **Sick** (RED): Symptomatic people who have the virus and are sick. Of those infected individuals, 50% get sick with the disease. In 10 days, a sick person may recover completely (98%) and develop lifelong immunity, or may die (2%).
- **Dead** (GRAY): People who die from the disease. 2% of the sick die.
- **Immune** (GREEN): There are two ways of getting to this state: 1) People who got sick with COVID-19, and recovered and thereby, have developed lifelong immunity, 2) People who were infected but didn't develop symptoms and became immune once they stopped being contagious.

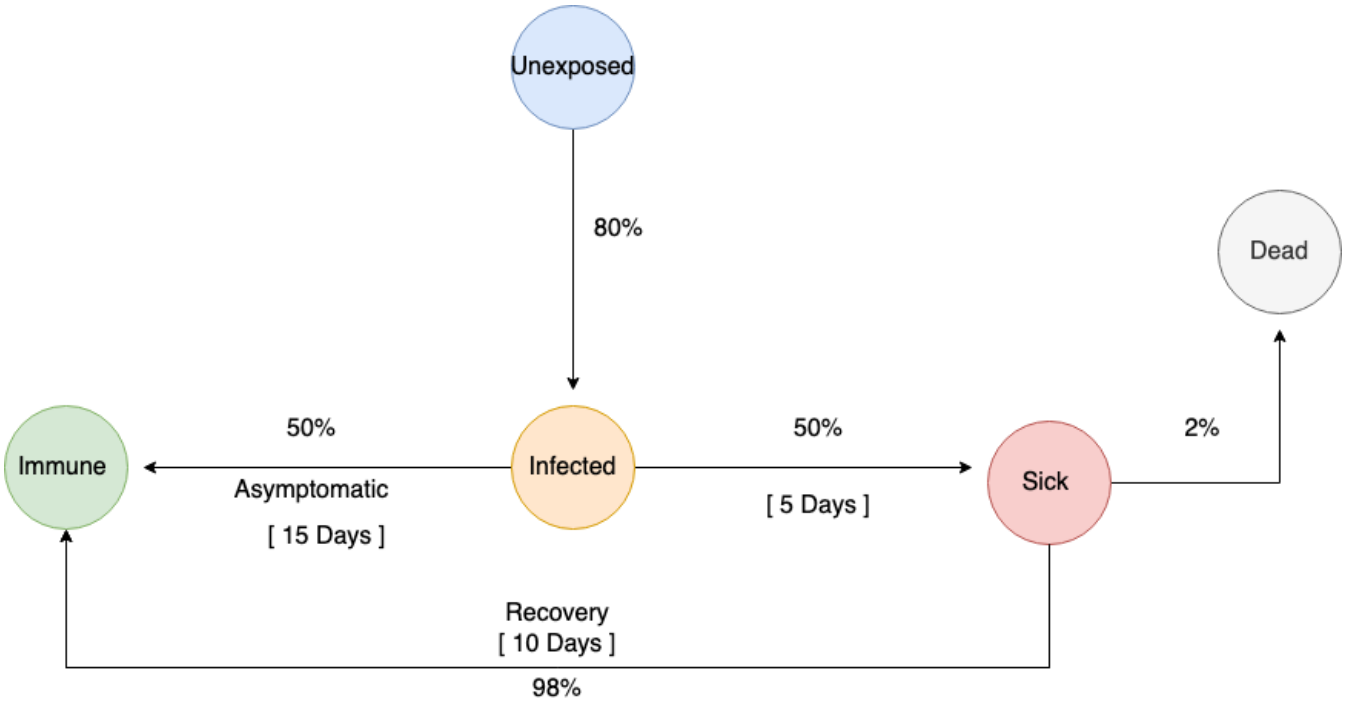
Let's assume that, the cost for running 1 day of simulation is 1 standard processing cycle of a specific computer. On day 1 of the simulation, a (random) person returned to the town from elsewhere and is sick with the virus. Unexposed people who come in contact with this carrier as they travel between home and work may become infected with the virus. There is an 80% chance of contracting the virus

(and changing state to infected) on contact with a carrier. We assume that this is the only means by which an unexposed person may get infected by the virus, and they are infected immediately. To summarize, the following transitions between states are possible:

- Unexposed → Infected (happens immediately on the same day after each contact with 80% chance)
- Infected → Sick (after 5 days of getting infected with 50% chance)
- Infected → Immune (after 15 days of getting infected with 50% chance)
- Sick → Immune (after 10 days of getting sick with 98% chance)
- Sick → Dead (after 10 days of getting sick with 2% chance)

Part 1

Draw the MDP graphically.



Part 2

Create the transition probabilities matrix/table for the MDP.

	Unexposed	Infected	Sick	Dead	Immune
Unexposed	0	0.8	0	0	0.2
Infected	0	0	0.5	0	0.5
Sick	0	0	0	0.02	0.98
Dead	0	0	0	1	0
Immune	0	0	0	0	1

Problem 4: MDP of a Professional Lifestyle

Solve the following MDP using both ***Value Iteration*** and ***Policy Iteration*** algorithms.

Neo is a freelance computer programmer, and his aim in life is to earn as much money as he can by writing more code. He has three possible states in his daily professional life as below.

1. **Productive** - He can write programming codes more efficiently.
2. **Exhausted** - He is too tired to think well in order to write efficient programming codes.
3. **Fit** - He is physically and mentally in a great state to write code and solve programming problems.

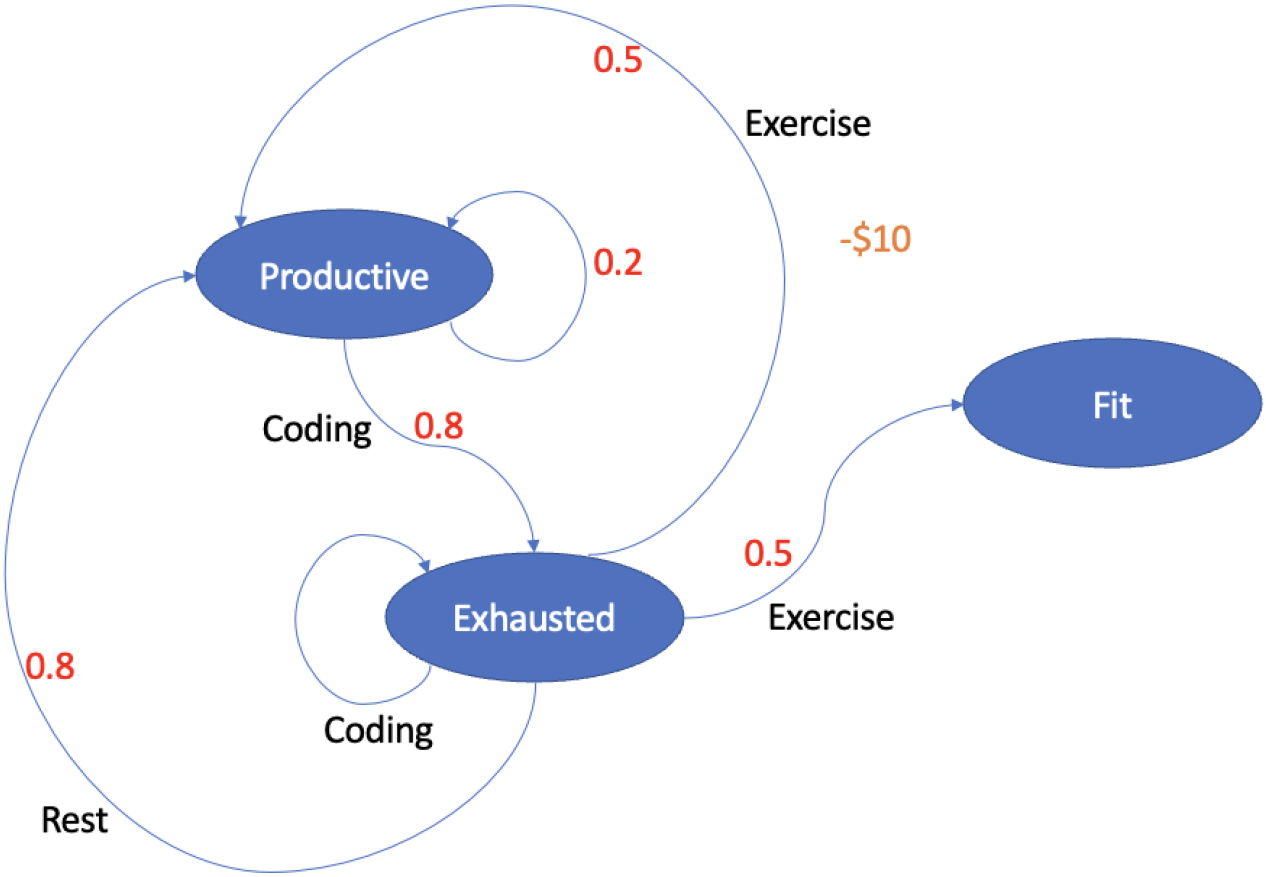
When Neo is **Exhausted**, he can choose one of three actions: (1) Keep **Coding**, (2) do some physical **Exercise**, or (3) get some **Rest**.

If he chooses to do more coding, he remains in the **Exhausted** state with the certainty of getting a $20 reward. If he decides to get Rest, he has 80% chance of moving to the next state, **Productive**, and a 20% chance of staying **Exhausted**. If he doesn't want to get Rest, he may go to the gym and do some physical Exercise. This gives him a 50% chance of entering the **Productive** state and a 50% chance of staying **Exhausted** state. However, he needs to pay for the gym, so this choice results in a -$10 reward.

When Neo becomes **Productive**, he can write programs more efficiently. From there, he has an 80% chance of getting **Exhausted** again with earning a $40 reward, and a 20% chance of staying **Productive** while earning a $30 reward. Sometimes, when he is **Productive**, he wants to do some physical Exercise. When he Exercises in this state, he enjoys it very much and gets 100% **Fit** physically and mentally. However, he needs to pay $10 for it.

Once Neo reaches the state **Fit**, he is fully-committed to earning more money by more Coding. Because he is in such a great state physically and mentally, he can write programs very efficiently. In this state, he earns a $100 reward, and he keeps writing code until he is **Exhausted** again.

Use MDP to find the best policy to maximize his earnings/rewards over time.



Part 1 Draw the MDP graphically.

Part 2

Using a discount factor of 0.86, solve the MDP using value iteration algorithm (until the values have become reasonably stable). You should start with the values set to zero. You should show both the optimal policy and the optimal values.

In [4]:
```
states = [0, 1, 2]  # Exhausted, Productive, Fit
actions = {
    0: ['coding', 'rest', 'exercise'],  # Exhausted actions
    1: ['coding', 'exercise'],  # Productive actions
    2: ['coding']  # Fit actions
}

rewards = {
    0: {'coding': 20, 'rest': 0, 'exercise': -10},
    1: {'coding': 30, 'exercise': 0},
    2: {'coding': 100}
}

transitions = {
    0: {'coding': [(0, 1.0)], 'rest': [(1, 0.8), (0, 0.2)], 'exercise': [(1, 0.5), (0, 0
    1: {'coding': [(0, 0.8), (1, 0.2)], 'exercise': [(2, 1.0)]},
    2: {'coding': [(0, 1.0)]}
}



def value_iteration(states, actions, rewards, transitions, gamma=0.9, theta=1e-5):
    V = {s: 0 for s in states}
```

```
    while True:
        delta = 0


        for s in states:
            v = V[s]


            Q = {}
            for a in actions[s]:
                q = rewards[s].get(a, 0)
                for s_next, p in transitions[s][a]:
                    q += gamma * p * V[s_next]
                Q[a] = q

            V[s] = max(Q.values())

            delta = max(delta, abs(v - V[s]))

        if delta < theta:
            break


    policy = {}
    for s in states:
        Q = {}
        for a in actions[s]:
            q = rewards[s].get(a, 0)
            for s_next, p in transitions[s][a]:
                q += gamma * p * V[s_next]
            Q[a] = q
        policy[s] = max(Q, key=Q.get)

    return V, policy



V, policy = value_iteration(states, actions, rewards, transitions)

print("Optimal value function:")
for s in states:
    print(f"V({s}) = {V[s]:.2f}")

print("Optimal policy:")
for s in states:
    print(f"Policy({s}) = {policy[s]}")
```

```
Optimal value function:
V(0) = 273.65
V(1) = 311.66
V(2) = 346.28
Optimal policy:
Policy(0) = rest
Policy(1) = exercise
Policy(2) = coding
```

Part 3

Using a discount factor of 0.86, solve the MDP using policy iteration algorithm (until you have complete convergence). You should start with the policy that always does **Coding**.

In [5]:

```
def policy_evaluation(pi, V, gamma, states, actions, rewards, transitions):
    theta = 1e-9  # A small positive number for convergence
    while True:
```

```
        delta = 0  # The maximum change of value across all states in this iteration
        for s in states:
            v = V[s]
            # Calculate the expected value under the current policy
            action = pi[s]
            expectation = 0
            for next_state, prob in transitions[s][action]:
                expectation += prob * (rewards[s][action] + gamma * V[next_state])
            V[s] = expectation
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break
    return V

def policy_improvement(pi, V, gamma, states, actions, rewards, transitions):
    policy_stable = True
    for s in states:
        old_action = pi[s]
        best_action = None
        best_value = float('-inf')
        for action in actions[s]:
            expectation = 0
            for next_state, prob in transitions[s][action]:
                expectation += prob * (rewards[s][action] + gamma * V[next_state])
            if expectation > best_value:
                best_action = action
                best_value = expectation
        pi[s] = best_action
        if old_action != best_action:
            policy_stable = False
    return pi, policy_stable


V = {0: 0, 1: 0, 2: 0}  # Value function for each state
pi = {0: 'coding', 1: 'coding', 2: 'coding'}  # Policy


gamma = 0.86
V = policy_evaluation(pi, V, gamma, states, actions, rewards, transitions)
pi, policy_stable = policy_improvement(pi, V, gamma, states, actions, rewards, transitio
```

Problem 5: Markov Decision Process (MDP) Toolbox for Python

One useful Python module for solving these types of problems is called *MDP toolbox* (pymdptoolbox). The MDP toolbox provides classes and functions for the resolution of descrete-time Markov Decision Processes. The list of algorithms that have been implemented includes backwards induction, linear programming, policy iteration, q-learning and value iteration along with several variations. In the next cell you'll see how to load this module into a Jupyter notebook running in Colab.

In [ ]:
```
!pip install pymdptoolbox
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publ
ic/simple/
Collecting pymdptoolbox
  Downloading pymdptoolbox-4.0-b3.zip (29 kB)
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.9/dist-packages (from pym
dptoolbox) (1.22.4)
Requirement already satisfied: scipy in /usr/local/lib/python3.9/dist-packages (from pym
dptoolbox) (1.10.1)
Building wheels for collected packages: pymdptoolbox
```

```
  Building wheel for pymdptoolbox (setup.py) ... done
  Created wheel for pymdptoolbox: filename=pymdptoolbox-4.0b3-py3-none-any.whl size=2565
5 sha256=7a1c7bfc3e8fb9ac8ded5b1771bf8ee6e088a52bfa36d031ae8d965b660c9d2a
  Stored in directory: /root/.cache/pip/wheels/e9/cd/97/2269d5ad0730978476c02fcb02383c82
055c99e1ede6ba15cd
Successfully built pymdptoolbox
Installing collected packages: pymdptoolbox
Successfully installed pymdptoolbox-4.0b3
```

A simple example code is given below. For details about the example, check the documentation for Python MDP Toolbox (pymdptoolbox) from this link.

In [ ]:
```
import numpy as np
import mdptoolbox
import mdptoolbox.example
np.random.seed(0) # Needed to get the output below
P, R = mdptoolbox.example.rand(2, 2)
pi = mdptoolbox.mdp.PolicyIteration(P, R, 0.9)
pi.run()
print(pi.policy)
print(pi.iter)
```

```
(1, 0)
1
```

Part 1

First read the documentation for Python MDP Toolbox (pymdptoolbox) from this link and try to understand how it works.

Then solve the MDP from *Problem 4* using both **value iteration** and **policy iteration** algorithms from the Python MDP Toolbox.

In [ ]:
```
# Defining the MDP problem
P = np.array([
    # Exhausted state
    [[0.8, 0.2, 0.0], [0.0, 0.5, 0.5], [0.8, 0.0, 0.2]],
    # Productive state
    [[0.2, 0.0, 0.8], [0.8, 0.0, 0.2], [0.0, 1.0, 0.0]],
    # Fit state
    [[1.0, 0.0, 0.0], [0.0, 0.8, 0.2], [0.0, 0.0, 1.0]]
])

R = np.array([
    # Exhausted state
    [20, -10, 0],
    # Productive state
    [30, 40, 0],
    # Fit state
    [100, 0, 0]
])

discount_factor = 0.95

mdpvi = mdptoolbox.mdp.ValueIteration(transitions=P, reward=R, discount=discount_factor)
mdppi = mdptoolbox.mdp.PolicyIteration(transitions=P, reward=R, discount=discount_factor

mdpvi.run()
mdppi.run()

print("Value iteration:")
print("Optimal value function: ", mdpvi.V)
print("Optimal policy: ", mdpvi.policy)
```

```
print("\nPolicy iteration:")
print("Optimal value function: ", mdppi.V)
print("Optimal policy: ", mdppi.policy)
```

```
Value iteration:
Optimal value function:  (563.8399129264144, 601.995840613409, 633.9037374272345)
Optimal policy:  (1, 0, 0)

Policy iteration:
Optimal value function:  (864.9681528662413, 903.1240521686375, 935.0318471337573)
Optimal policy:  (1, 0, 0)
```