

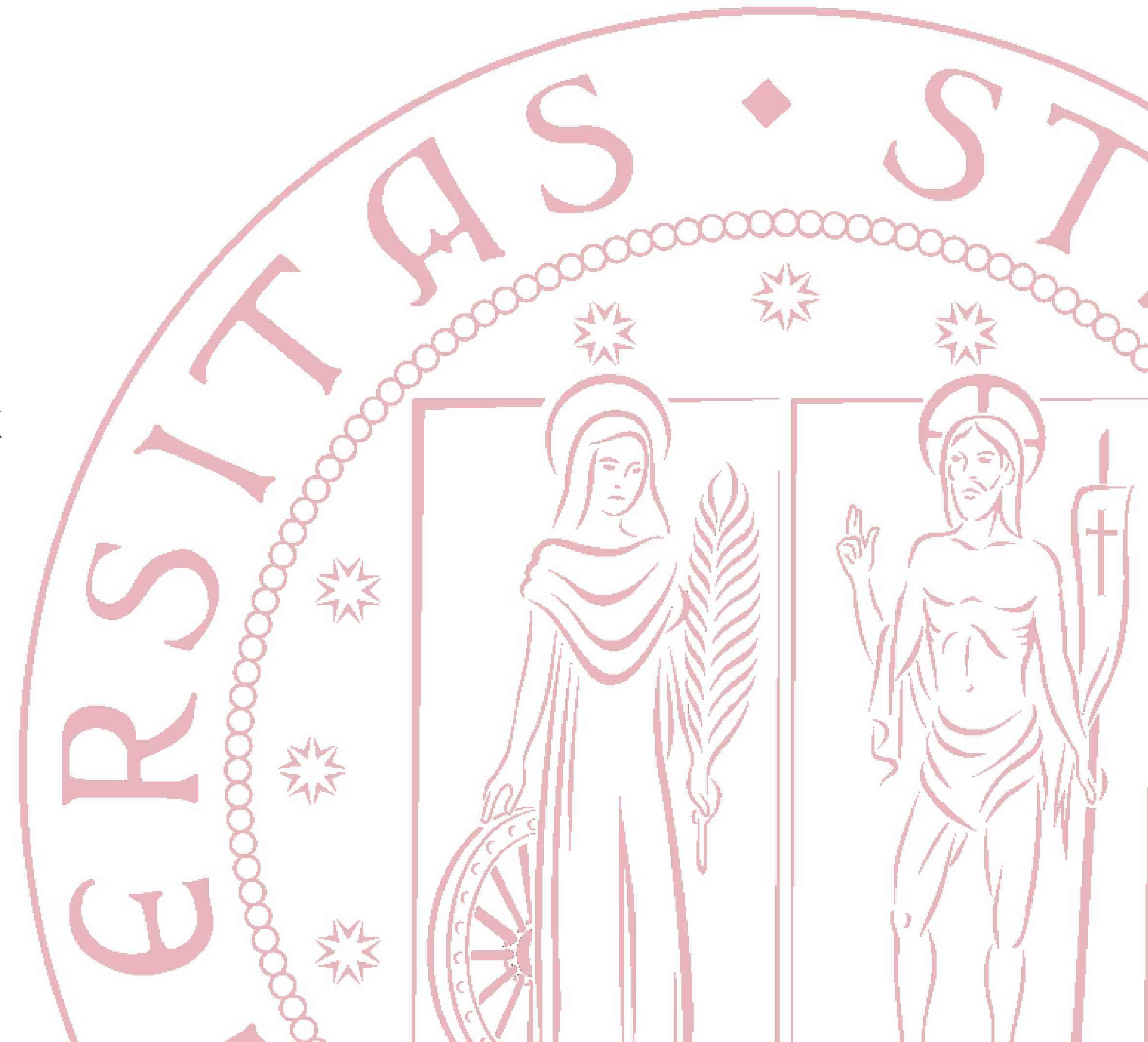
Markov Decision Processes in Python

Gloria Beraldo (gloria.beraldo@unipd.it)

Department of Information Engineering, University of Padova

Topics:

- Utility recap
- Example: Time for coffee?
- Example: Time for coffee?- Maximax – Maximin
- MDP toolbox
- Exercise: Solving a simple MDP using the MDP toolbox
- Value iteration algorithm recap
- Policy iteration algorithm recap
- Value iteration algorithm via MDP toolbox
- Policy iteration algorithm via MDP toolbox
- Q-Learning via MDP toolbox



Expected utility: Recap

Expected utility

- Let's call $U(s)$ the **utility** of being in a certain state s
- $U(s)$ is a sort of numerical “score” assigned to s – i.e. how desirable is to reach that particular state (e.g. money earned in the quiz game)
- The **expected utility** of an action, $EU(a)$, is then the average utility value of the outcomes, weighted by the probability that the outcome occurs:

$$EU(a) = \sum_{s'} P(\text{RESULT}(a) = s') U(s')$$

- **MEU principle**: choose the action that maximizes expected utility

Expected utility: Example Time for coffee?

I have half an hour to spare in my busy schedule, and I have a choice between working quietly in my office and going out for a coffee.

If I stay in my office, three things can happen:

- I can get some **work done** (**Utility = 8**),
- I can **get distracted** looking at the latest news (**Utility = 1**),
- A **colleague might stop** by to talk about some work we are doing (**Utility = 5**).

If I go out for coffee, I will most likely enjoy a good cup of **smooth caffeination** (**Utility = 10**), but there is also a chance I will end up **spilling coffee** all over myself (**Utility = -20**).

The probability of getting work done if I choose to **stay in the office** is **0.5**, while the probabilities of **getting distracted**, and a **colleague stopping** by are **0.3** and **0.2** respectively.

If I go out for a coffee, my chance of **enjoying my beverage** is **0.95**, and the chance of **spilling my drink** is **0.05**.

QUESTION 1(a): What is the expected utility of staying in my office?

QUESTION 1(b): What is the expected utility of going out for a coffee?

QUESTION 1(c): By the principle of maximum expected utility, what should I do?



Expected utility: Example Time for coffee?

QUESTION 1(a): What is the expected utility of staying in my office?

SOLUTION 1(a):

Staying in the *office* means that I will either *work*, get *distracted*, or talk with a *colleague*. These states have the following utilities:

$$U(\textit{work}) = 8$$

$$U(\textit{distracted}) = 1$$

$$U(\textit{colleague}) = 5$$

and the probabilities of these happening, given I stay in the *office* are:

$$P(\textit{work}/\textit{office}) = 0.5$$

$$P(\textit{distracted}/\textit{office}) = 0.3$$

$$P(\textit{colleague}/\textit{office}) = 0.2$$

Let's declare these as Python arrays



Expected utility: Example Time for coffee?

Let's declare these as Python arrays

```
import numpy as np
# Setup arrays with: symbolic names for outcomes (not currently used), utilities of outcomes, and
# probabilities of those outcomes
office_outcomes = ["work", "distracted", "colleague"]
print('office outcomes = ', office_outcomes)
u_office_outcomes = np.array([8, 1, 5])
print('U(office_outcomes) = ', u_office_outcomes)
p_office_outcomes_office = np.array([0.5, 0.3, 0.2])
print('P(office_outcomes|office) = ', p_office_outcomes_office)
```

```
office_outcomes = ['work', 'distracted', 'colleague']
U(office_outcomes) = [8 1 5]
P(office_outcomes|office) = [0.5 0.3 0.2]
```

Expected utility: Example Time for coffee?

The **expected utility** of an action, $EU(a)$, is then the average utility value of the outcomes, weighted by the probability that the outcome occurs:

$$EU(a) = \sum_{s'} P(\text{RESULT}(a) = s') U(s')$$

SOLUTION 1(a):

$$\begin{aligned} EU(\text{office}) = & P(\text{work/office}) * U(\text{work}) + \\ & P(\text{distracted/office}) * U(\text{distracted}) + \\ & P(\text{colleague/office}) * U(\text{colleague}) \end{aligned}$$

$$EU(\text{office}) = 0.5 * 8 + 0.3 * 1 + 0.2 * 5 = 5.3$$

Expected utility: Example Time for coffee?

SOLUTION 1(a):

Now let's implement this in Python:

```
# The weighted utility of each outcome is each to compute by pairwise multiplication
eu_office_outcomes = u_office_outcomes * p_office_outcomes_office
print('EU by outcome =', eu_office_outcomes)
# Summing the weighted utilities gets us the expected utility
eu_office = np.sum(eu_office_outcomes)
print('EU(office) = ', eu_office)
```

```
EU by outcome = [4.  0.3 1. ]
EU(office) =  5.3
```

So the expected utility of staying in the office is **5.3**



Expected utility: Example Time for coffee?

QUESTION 1(b): What is the expected utility of going out for a coffee?

SOLUTION 1(b):

Going out for a coffee means that I will either *enjoying* my beverage, or *spilling* my drink. These states have the following utilities:

$$U(\textit{caffeination}) = 10$$

$$U(\textit{spillage}) = -20$$

and the probabilities of these happening, given I stay in the *office* are:

$$P(\textit{caffeination}/\textit{coffee}) = 0.95$$

$$P(\textit{spillage}/\textit{coffee}) = 0.05$$

Let's implement these in Python using the same notation as before



Expected utility: Example Time for coffee?

SOLUTION 1(b):

```
# The coffee calculation is the same as the office calculation, first set up arrays
coffee_outcomes = ["caffeination", "spillage"]
print('coffee_outcomes = ', coffee_outcomes)
u_coffee_outcomes = np.array([10, -20])
print('U(coffee_outcomes) = ', u_coffee_outcomes)
p_coffee_outcomes_coffee = np.array([0.95, 0.05])
print('P(coffee_outcomes|coffee) =', p_coffee_outcomes_coffee)
print('\n')
# Then compute the expected utility
eu_coffee_outcomes = u_coffee_outcomes * p_coffee_outcomes_coffee
print('EU by outcome =', eu_coffee_outcomes)
eu_coffee = np.sum(eu_coffee_outcomes)
print('EU(coffee) = ', eu_coffee)
```

```
coffee_outcomes = ['caffeination', 'spillage']
U(coffee_outcomes) = [ 10 -20]
P(coffee_outcomes|coffee) = [0.95 0.05]
```

```
EU by outcome = [ 9.5 -1. ]
EU(coffee) = 8.5
```

So the expected utility of going out for coffee is **8.5**



Expected utility: Example Time for coffee?

QUESTION 1(c): By the principle of maximum expected utility, what should I do?

SOLUTION 1(c):

MEU principle: choose the action that maximizes expected utility

Clearly in the case of numbers in the example, the option of *going out for coffee* is the one with the maximum expected utility.

However, we will also program it in Python so that we can see what happens as the probabilities of the outcomes vary:

```
if eu_office > eu_coffee:  
    print('Office is the MEU choice')  
else:  
    print('Coffee is the MEU choice')
```

Coffee is the MEU choice



Expected utility: Example Time for coffee? - Maximax

Revisit the decision for the coffee example using the **maximax decision criterion**

SOLUTION:

The **maximax decision criterion** rates each choice by the **utility** of its **best outcome**, and then picks the choice with best utility.

In Python we would do this calculation as follows

```
# The utility of each choice is the max utility of their outcomes
max_u_office = np.max(u_office_outcomes)
print('MaxU(office) =', max_u_office)
max_u_coffee = np.max(u_coffee_outcomes)
print('MaxU(coffee) =', max_u_coffee)
print('\n')
# The decision criterion is then to pick the outcome with the highest utility:
if max_u_office > max_u_coffee:
    print('Office is the Maximax choice')
else:
    print('Coffee is the Maximax choice')
```

```
MaxU(office) = 8
MaxU(coffee) = 10
```

```
Coffee is the Maximax choice
```



Expected utility: Example Time for coffee? - Maximin

Revisit the decision for the coffee example using the **maximin decision criterion**

SOLUTION:

The **maximin decision criterion** rates each choice by the **utility of its worst outcome**, and then picks the choice with the best utility.

In Python we would do this calculation as follows

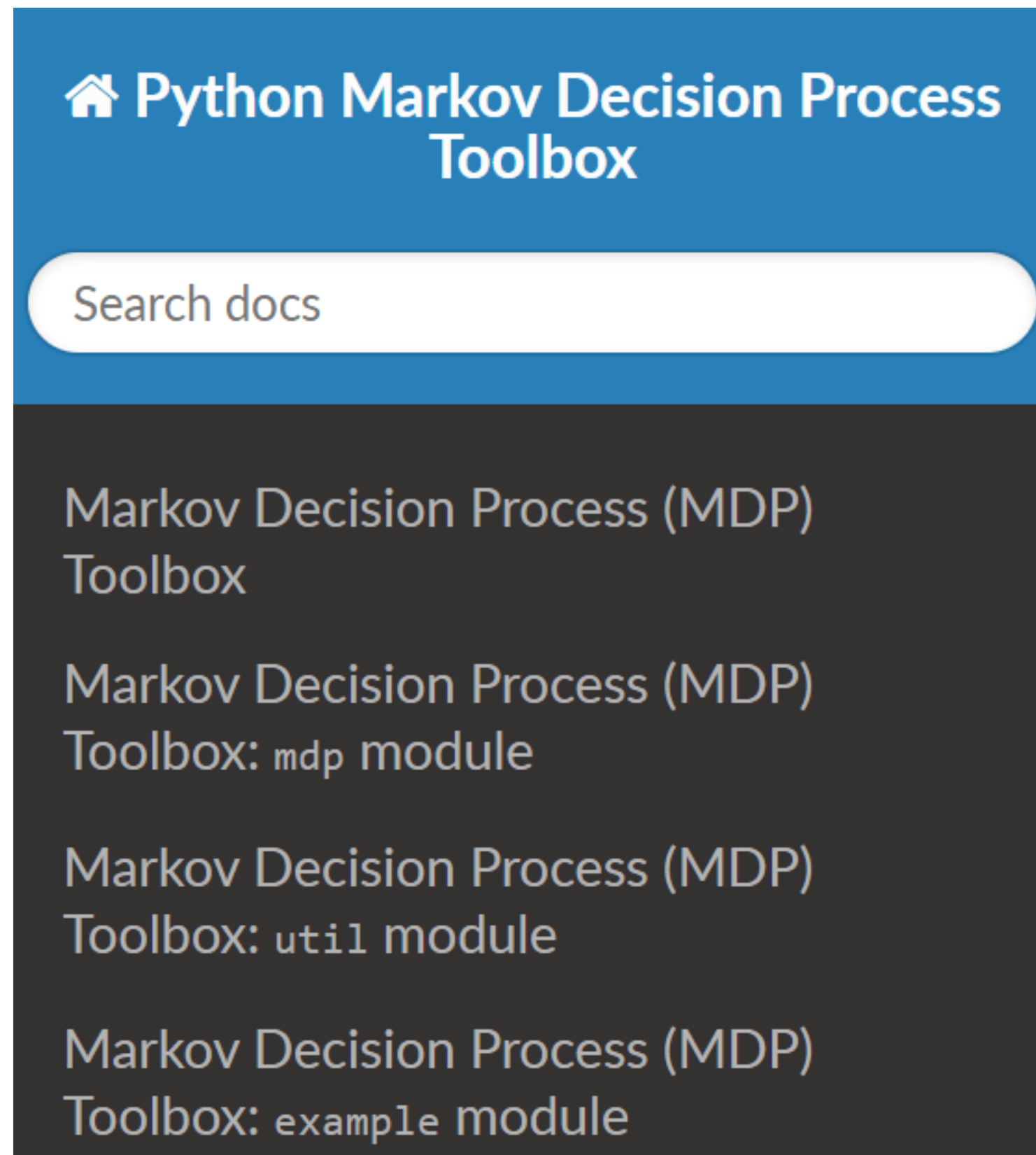
```
# The utility of each choice is the max utility of their outcomes
min_u_office = np.min(u_office_outcomes)
print('MinU(office) =', min_u_office)
min_u_coffee = np.min(u_coffee_outcomes)
print('MinU(coffee) =', min_u_coffee)
print('\n')
# The decision criterion is then to pick the outcome with the highest utility:
if min_u_office > min_u_coffee:
    print('Office is the Minimax choice')
else:
    print('Coffee is the Minimax choice')
```

```
MinU(office) = 1
MinU(coffee) = -20
```

Office is the Minimax choice



MDP toolbox



The MDP toolbox provides classes and functions for the resolution of **discrete-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.

The documentation is available at:

<https://pymdptoolbox.readthedocs.io/en/latest/index.html>

We need to install it using:

```
pip install pymdptoolbox
```

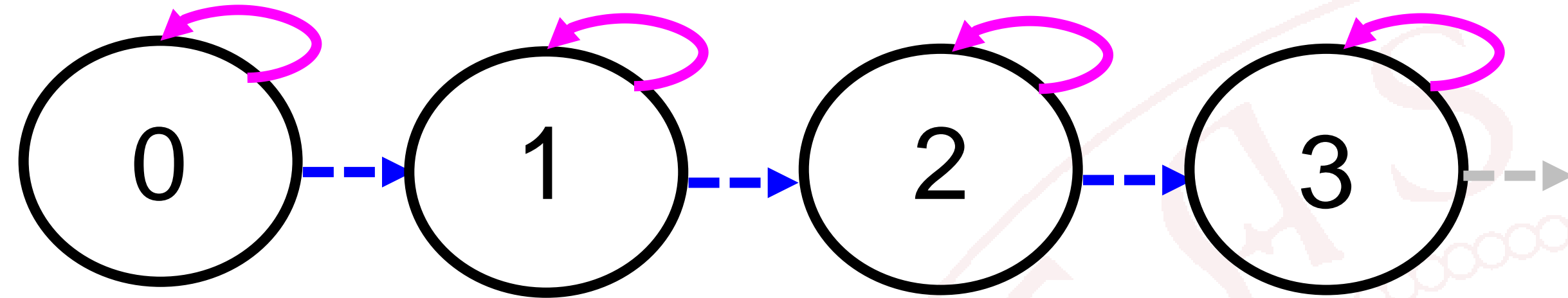
Exercise: Solving a simple MDP using the MDP toolbox

Let's start with a really simple problem.

We have 4 states and two actions.

There are two actions:

- 0 is "Stay",
- 1 is "Right".



0 **always succeeds** and leaves the agent in the same state.

1 moves the agent right with **probability 0.8**, stays in place with **probability 0.2**.

The states are 0, 1, 2, 3.

0 is left of 1, which is left of 2 and so on.

(Thus the states are in a line which runs 0, 1, 2, 3 from left to right.)

The agent remains in **state 3** with **probability 1**.

State 3 has a **reward of 1**, and the **cost** of any action is **-0.04**.

Exercise: Solving a simple MDP using the MDP toolbox

The MDP Toolbox defines MDPs through a **probability array** and a **reward array**.

The probability array has shape **(A, S, S)**, where **A** are **actions** and **S** are **states**.

For **each action** specify the **transitions probabilities** of reaching the **second state** by applying that action in the **first state**.

```
!pip install pymdptoolbox
import mdptoolbox
import numpy as np

# The MDP Toolbox defines MDPs through a probability array and a reward array.

# The probability array has shape (A, S, S), where A are actions and S
# are states. For each action specify the transitions probabilities of reaching
# the second state by applying that action in the first state.

# So, to implement the action model described above, we need:
P1 = np.array([[[[1, 0, 0, 0],
                  [0, 1, 0, 0],
                  [0, 0, 1, 0],
                  [0, 0, 0, 1]],
                 [[0.2, 0.8, 0, 0],
                  [0, 0.2, 0.8, 0],
                  [0, 0, 0.2, 0.8],
                  [0, 0, 0, 1]]]])
```


Exercise: Solving a simple MDP using the MDP toolbox

The MDP Toolbox defines MDPs through a **probability array** and a **reward array**.

The probability array has shape **(A, S, S)**, where **A** are **actions** and **S** are **states**.

For **each action** specify the **transitions probabilities** of reaching the **second state** by applying that action in the **first state**.

```
!pip install pymdptoolbox
```

```
import mdptoolbox
```

```
import numpy as np
```

```
# The MDP Toolbox defines MDPs through a probability array and a reward array.
```

```
# The probability array has shape (A, S, S), where A are actions and S  
# are states. For each action specify the transitions probabilities of reaching  
# the second state by applying that action in the first state.
```

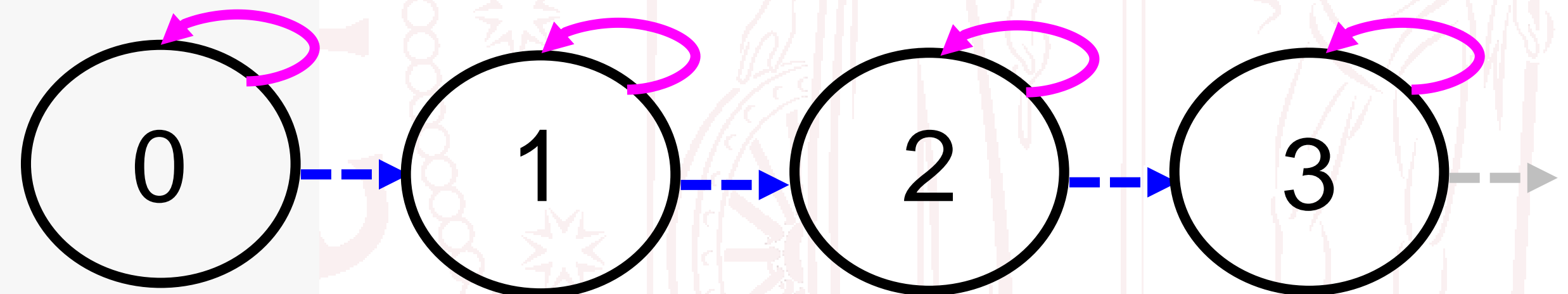
```
# So, to implement the action model described above, we need:
```

```
P1 = np.array([[[1, 0, 0, 0],  
                [0, 1, 0, 0],  
                [0, 0, 1, 0],  
                [0, 0, 0, 1]],
```

Stay

```
                [[0.2, 0.8, 0, 0],  
                [0, 0.2, 0.8, 0],  
                [0, 0, 0.2, 0.8],  
                [0, 0, 0, 1]]])
```

Right



Exercise: Solving a simple MDP using the MDP toolbox

The MDP Toolbox defines MDPs through a **probability array** and a **reward array**.

The probability array has shape **(A, S, S)**, where **A** are **actions** and **S** are **states**.

For **each action** specify the **transitions probabilities** of reaching the **second state** by applying that action in the **first state**.

```
!pip install pymdptoolbox
import mdptoolbox
import numpy as np

# The MDP Toolbox defines MDPs through a probability array and a reward array.

# The probability array has shape (A, S, S), where A are actions and S
# are states. For each action specify the transitions probabilities of reaching
# the second state by applying that action in the first state.

# So, to implement the action model described above, we need:
P1 = np.array([[1, 0, 0, 0],
               [0, 1, 0, 0],
               [0, 0, 1, 0],
               [0, 0, 0, 1]],
               [[0.2, 0.8, 0, 0],
                [0, 0.2, 0.8, 0],
                [0, 0, 0.2, 0.8],
                [0, 0, 0, 1]])
```

Stay

Right

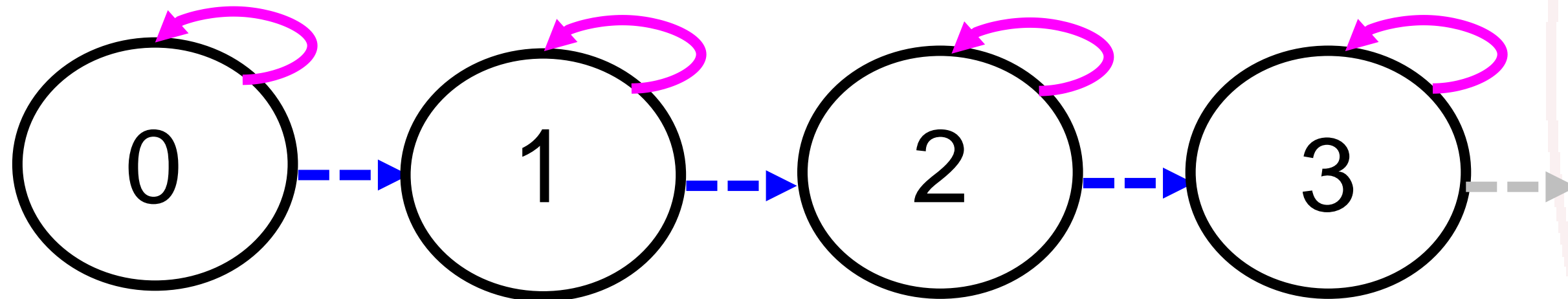
The **first matrix** is that for the action "Stay" (when executed in a given state the agent stays there) and the **second** is for the action "Right" (which shifts the agent right with probability 0.8 except in state 3 when the agent remains in state 3 with probability 1).

Exercise: Solving a simple MDP using the MDP toolbox

The reward array has **shape (S, A)**, so there is a set of S vectors, one for each state, and each is a vector with one element for each the actions --- **each element is the reward for executing the relevant action in the state** (so this is really modelling cost of the action).

```
R1 = np.array([[-0.04, -0.04], [-0.04, -0.04], [-0.04, -0.04], [1, 1]])  
# R1 says that executing either action in states 0, 1, or 2 has a reward  
# of -0.04, and executing either action in state 3 has reward 1.
```

R1 says that executing either action in states 0, 1, or 2 has a **reward of -0.04**, and executing either action in **state 3 has reward 1**.



Exercise: Solving a simple MDP using the MDP toolbox

The `util.check()` function checks that the reward and probability matrices are well-formed, and match. Success is silent, failure provides somewhat useful error messages.

```
# The util.check() function checks that the reward and probability matrices
# are well-formed, and match.
#
# Success is silent, failure provides somewhat useful error messages.
mdptoolbox.util.check(P1, R1)
```

`mdptoolbox.util.check(P, R)` [\[source\]](#)

Check if `P` and `R` define a valid Markov Decision Process (MDP).

Let `S` = number of states, `A` = number of actions.

Parameters:

- `P (array)` - **The transition matrices** It can be a three dimensional array with a shape of (A, S, S) . It can also be a one dimensional array with a shape of $(A,)$, where each element contains a matrix of shape (S, S) which can possibly be sparse.
- `R (array)` - **The reward matrix** It can be a three dimensional array with a shape of (S, A, A) . It can also be a one dimensional array with a shape of $(A,)$, where each element contains matrix with a shape of (S, S) which can possibly be sparse. It can also be an array with a shape of (S, A) which can possibly be sparse.

Notes

Raises an error if `P` and `R` do not define a MDP.

<https://pymdptoolbox.readthedocs.io/en/latest/api/util.html>

Value iteration algorithm recap



Value iteration algorithm

- The iteration step, called a Bellman update, looks like this
- Update is assumed to be applied simultaneously to all the states at each iteration, where

function Q-VALUE(*mdp*, *s*, *a*, *U*) **returns** a utility value
return $\sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U[s']]$

- The algorithm stops when the difference between old and updated utilities is below a certain threshold

$$U_{i+1}(s) \leftarrow \max_{a \in A(s)} \sum_{s'} P(s' | s, a) [R(s, a, s') + \gamma U_i(s')],$$

```
function VALUE-ITERATION(mdp,  $\epsilon$ ) returns a utility function
inputs: mdp, an MDP with states S, actions A(s), transition model  $P(s' | s, a)$ ,
        rewards  $R(s, a, s')$ , discount  $\gamma$ 
         $\epsilon$ , the maximum error allowed in the utility of any state
local variables: U, U', vectors of utilities for states in S, initially zero
         $\delta$ , the maximum relative change in the utility of any state

repeat
     $U \leftarrow U'$ ;  $\delta \leftarrow 0$ 
    for each state s in S do
         $U'[s] \leftarrow \max_{a \in A(s)} \text{Q-VALUE}(\textit{mdp}, s, a, U)$ 
        if  $|U'[s] - U[s]| > \delta$  then  $\delta \leftarrow |U'[s] - U[s]|$ 
until  $\delta \leq \epsilon(1 - \gamma)/\gamma$ 
return U
```


Value iteration algorithm via MDP toolbox

```
class mdptoolbox.mdp.ValueIteration(transitions, reward, discount, epsilon=0.01, max_iter=1000,
initial_value=0, skip_check=False) [source]
```

Bases: `mdptoolbox.mdp.MDP`

A discounted MDP solved using the value iteration algorithm.

ValueIteration applies the value iteration algorithm to solve a discounted MDP. The algorithm consists of solving Bellman's equation iteratively. Iteration is stopped when an epsilon-optimal policy is found or after a specified number (`max_iter`) of iterations. This function uses verbose and silent modes. In verbose mode, the function displays the variation of `v` (the value function) for each iteration and the condition which stopped the iteration: epsilon-policy found or maximum number of iterations reached.

Parameters:

- **transitions** (*array*) – transition probability matrices. See the documentation for the `MDP` class for details.
- **reward** (*array*) – Reward matrices or vectors. See the documentation for the `MDP` class for details.
- **discount** (*float*) – Discount factor. See the documentation for the `MDP` class for details.
- **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the `MDP` class for details. Default: 0.01.
- **max_iter** (*int, optional*) – Maximum number of iterations. If the value given is greater than a computed bound, a warning informs that the computed bound will be used instead. By default, if `discount` is not equal to 1, a bound for `max_iter` is computed, otherwise `max_iter` = 1000. See the documentation for the `MDP` class for further details.
- **initial_value** (*array, optional*) – The starting value function. Default: a vector of zeros.
- **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
- **Attributes** (*Data*) –
 - `-----`
 - **V** (*tuple*) – The optimal value function.
 - **policy** (*tuple*) – The optimal policy function. Each element is an integer corresponding to an action which maximises the value function in that state.
 - **iter** (*int*) – The number of iterations taken to complete the computation.
 - **time** (*float*) – The amount of CPU time used to run the algorithm.

`run()` [source]

Do the algorithm iteration.

To run value iteration we create a value iteration object, and run it.

Note that the discount value is 0.9

```
# To run value iteration we create a value iteration object, and run it. Note that
# discount value is 0.9
vi1 = mdptoolbox.mdp.ValueIteration(P1, R1, 0.9)
vi1.run()
```

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Exercise: Solving a simple MDP using the MDP toolbox

```
class mdptoolbox.mdp.ValueIteration(transitions, reward, discount, epsilon=0.01, max_iter=1000,
initial_value=0, skip_check=False) [source]
```

Bases: mdptoolbox.mdp.MDP

A discounted MDP solved using the value iteration algorithm.

ValueIteration applies the value iteration algorithm to solve a discounted MDP. The algorithm consists of solving Bellman's equation iteratively. Iteration is stopped when an epsilon-optimal policy is found or after a specified number (`max_iter`) of iterations. This function uses verbose and silent modes. In verbose mode, the function displays the variation of `v` (the value function) for each iteration and the condition which stopped the iteration: epsilon-policy found or maximum number of iterations reached.

- Parameters:
- **transitions** (*array*) – Transition probability matrices. See the documentation for the `MDP` class for details.
 - **reward** (*array*) – Reward matrices or vectors. See the documentation for the `MDP` class for details.
 - **discount** (*float*) – Discount factor. See the documentation for the `MDP` class for details.
 - **epsilon** (*float, optional*) – Stopping criterion. See the documentation for the `MDP` class for details. Default: 0.01.
 - **max_iter** (*int, optional*) – Maximum number of iterations. If the value given is greater than a computed bound, a warning informs that the computed bound will be used instead. By default, if `discount` is not equal to 1, a bound for `max_iter` is computed, otherwise `max_iter` = 1000. See the documentation for the `MDP` class for further details.
 - **initial_value** (*array, optional*) – The starting value function. Default: a vector of zeros.
 - **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
 - **Attributes (Data)** –
 - **-----**
 - **V** (*tuple*) – The optimal value function.
 - **policy** (*tuple*) – The optimal policy function. Each element is an integer corresponding to an action which maximises the value function in that state.
 - **iter** (*int*) – The number of iterations taken to complete the computation.
 - **time** (*float*) – The amount of CPU time used to run the algorithm.

We can then display the values (utilities) computed, and look at the policy:

```
# We can then display the values (utilities) computed, and look at the policy:
print('Values:\n', vi1.V)
print('Policy:\n', vi1.policy)
```

Values:
(2.766226988084275, 3.7438891127976524, 4.857502678650809, 6.12579511)
Policy:
(1, 1, 1, 0)

This says that the optimum policy is to go Right in every state until reaching state 3, then Stay.

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Policy iteration algorithm recap



Policy iteration algorithm

- How to do **POLICY-EVALUATION**?
- Action in each state is fixed by the policy

- At the i^{th} iteration, the policy π_i specifies the action $\pi_i(s)$ in state s

$$U_i(s) = \sum_{s'} P(s' | s, \pi_i(s)) [R(s, \pi_i(s), s') + \gamma U_i(s')].$$

- Basically a simplified version of Bellman eq. relating the utility of s (with π_i) to those of its neighbors
 - No “**max**” operator \Rightarrow linear equations

```
function POLICY-ITERATION(mdp) returns a policy
  inputs: mdp, an MDP with states  $S$ , actions  $A(s)$ , transition model  $P(s' | s, a)$ 
  local variables:  $U$ , a vector of utilities for states in  $S$ , initially zero
                   $\pi$ , a policy vector indexed by state, initially random

  repeat
     $U \leftarrow \text{POLICY-EVALUATION}(\pi, U, \textit{mdp})$ 
    unchanged?  $\leftarrow$  true
    for each state  $s$  in  $S$  do
       $a^* \leftarrow \underset{a \in A(s)}{\text{argmax}} \text{ Q-VALUE}(\textit{mdp}, s, a, U)$ 
      if  $\text{Q-VALUE}(\textit{mdp}, s, a^*, U) > \text{Q-VALUE}(\textit{mdp}, s, \pi[s], U)$  then
         $\pi[s] \leftarrow a^*$ ; unchanged?  $\leftarrow$  false
  until unchanged?
  return  $\pi$ 
```


Policy iteration algorithm via MDP toolbox

```
class mdptoolbox.mdp.PolicyIteration(transitions, reward, discount, policy0=None, max_iter=1000, eval_type=0, skip_check=False) [source]
```

Bases: `mdptoolbox.mdp.MDP`

A discounted MDP solved using the policy iteration algorithm.

- Parameters:
- **transitions** (*array*) – Transition probability matrices. See the documentation for the `MDP` class for details.
 - **reward** (*array*) – Reward matrices or vectors. See the documentation for the `MDP` class for details.
 - **discount** (*float*) – Discount factor. See the documentation for the `MDP` class for details.
 - **policy0** (*array, optional*) – Starting policy.
 - **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the `MDP` class for details. Default is 1000.
 - **eval_type** (*int or string, optional*) – Type of function used to evaluate policy. 0 or “matrix” to solve as a set of linear equations. 1 or “iterative” to solve iteratively. Default: 0.
 - **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
 - **Attributes (Data)** –
 - `-----`
 - **V** (*tuple*) – value function
 - **policy** (*tuple*) – optimal policy
 - **iter** (*int*) – number of done iterations
 - **time** (*float*) – used CPU time

Although we have been looking at the policy, we go it through value iteration.

Solving the same problem using policy iteration is easy with the MDP Toolbox:

```
# To run policy iteration we create a policy iteration object, and run it. Note that  
# discount value is 0.9  
pi1 = mdptoolbox.mdp.PolicyIteration(P1, R1, 0.9)  
pi1.run()
```

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Exercise: Solving a simple MDP using the MDP toolbox

```
class mdptoolbox.mdp.PolicyIteration(transitions, reward, discount, policy0=None, max_iter=1000, eval_type=0, skip_check=False) [source]
```

Bases: `mdptoolbox.mdp.MDP`

A discounted MDP solved using the policy iteration algorithm.

- Parameters:
- **transitions** (*array*) – Transition probability matrices. See the documentation for the `MDP` class for details.
 - **reward** (*array*) – Reward matrices or vectors. See the documentation for the `MDP` class for details.
 - **discount** (*float*) – Discount factor. See the documentation for the `MDP` class for details.
 - **policy0** (*array, optional*) – Starting policy.
 - **max_iter** (*int, optional*) – Maximum number of iterations. See the documentation for the `MDP` class for details. Default is 1000.
 - **eval_type** (*int or string, optional*) – Type of function used to evaluate policy. 0 or “matrix” to solve as a set of linear equations. 1 or “iterative” to solve iteratively. Default: 0.
 - **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
 - **Attributes (Data)** –
 - -----
 - **V** (*tuple*) – value function
 - **policy** (*tuple*) – optimal policy
 - **iter** (*int*) – number of done iterations
 - **time** (*float*) – used CPU time

Although we have been looking at the policy, we go it through value iteration.

Solving the same problem using policy iteration is easy with the MDP Toolbox:

```
# We can then display the values (utilities) computed, and look at the policy:
print('Values:\n', pi1.V)
print('Policy:\n', pi1.policy)
```

```
Values:
(6.6402692938291725, 7.6180844735276665, 8.731707317073173, 10.000000000000002)
Policy:
(1, 1, 1, 0)
```

Note that the methods disagree on the value while agreeing on the policy.

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Q-Learning algorithm via MDP toolbox

```
class mdptoolbox.mdp.QLearning(transitions, reward, discount, n_iter=10000, skip_check=False) [source]
```

Bases: `mdptoolbox.mdp.MDP`

A discounted MDP solved using the Q learning algorithm.

- Parameters:
- **transitions** (*array*) - Transition probability matrices. See the documentation for the `MDP` class for details.
 - **reward** (*array*) - Reward matrices or vectors. See the documentation for the `MDP` class for details.
 - **discount** (*float*) - Discount factor. See the documentation for the `MDP` class for details.
 - **n_iter** (*int, optional*) - Number of iterations to execute. This is ignored unless it is an integer greater than the default value. Default: 10,000.
 - **skip_check** (*bool*) - By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
 - **Attributes (Data)** -
 - -----
 - **Q** (*array*) - learned Q matrix (SxA)
 - **V** (*tuple*) - learned value function (S).
 - **policy** (*tuple*) - learned optimal policy (S).
 - **mean_discrepancy** (*array*) - Vector of V discrepancy mean over 100 iterations. Then the length of this vector for the default value of N is 100 (N/100).

Solving a problem using reinforcement learning (well, the **Q-learning** kind of Reinforcement Learning)

Action-utility function, or Q-function: $Q(s, a)$

- expected utility of taking a given action in a given state
- related to utilities in the obvious way →

is also easy using the MDP Toolbox:

```
# To run q-learning we create a q-learning object, and run it. Note that
# discount value is 0.9
ql1 = mdptoolbox.mdp.QLearning(P1, R1, 0.9)
ql1.run()
```

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Exercise: Solving a simple MDP using the MDP toolbox

```
class mdptoolbox.mdp.QLearning(transitions, reward, discount, n_iter=10000, skip_check=False) [source]
```

Bases: `mdptoolbox.mdp.MDP`

A discounted MDP solved using the Q learning algorithm.

- Parameters:
- **transitions** (*array*) – Transition probability matrices. See the documentation for the `MDP` class for details.
 - **reward** (*array*) – Reward matrices or vectors. See the documentation for the `MDP` class for details.
 - **discount** (*float*) – Discount factor. See the documentation for the `MDP` class for details.
 - **n_iter** (*int, optional*) – Number of iterations to execute. This is ignored unless it is an integer greater than the default value. Default: 10,000.
 - **skip_check** (*bool*) – By default we run a check on the `transitions` and `rewards` arguments to make sure they describe a valid MDP. You can set this argument to True in order to skip this check.
 - **Attributes (Data) –**
 - -----
 - **Q** (*array*) – learned Q matrix (SxA)
 - **V** (*tuple*) – learned value function (S).
 - **policy** (*tuple*) – learned optimal policy (S).
 - **mean_discrepancy** (*array*) – Vector of V discrepancy mean over 100 iterations. Then the length of this vector for the default value of N is 100 (N/100).

Solving a problem using reinforcement learning (well, the **Q-learning** kind of Reinforcement Learning) is also easy using the MDP Toolbox:

```
# We can then display the values (utilities) computed, and look at the policy:
print('Values:\n', ql1.V)
print('Policy:\n', ql1.policy)
```

```
Values:
(0.26457661943006405, 2.0695643950241327, 6.421037194032352, 9.999999762119485)
Policy:
(1, 1, 1, 0)
```

Note that the methods disagree on the value while agreeing on the policy.

<https://pymdptoolbox.readthedocs.io/en/latest/api/mdp.html>

Questions

