

# THE AGENT WHO AVOIDED THE ORDINARY

In this lab, you will implement the simplest agent that learns to predict the outcome of its actions and tries another action when it gets bored.

## Learning objective

Upon completing this lab, you will be able to implement artificial agents based on the 'conceptual inversion of the interaction cycle.' In this framework, the agent starts by taking action and then receives a sensory signal which is an outcome of action. This contrasts with traditional AI agents, which first perceive their environment before deciding how to act.

## Setup

### Define the Agent class

```
In [11]: class Agent:
    def __init__(self):
        """ Creating our agent """
        self._action = None
        self._predicted_outcome = None

    def action(self, _outcome):
        """ tracing the previous cycle """
        if self._action is not None:
            print(f"Action: {self._action}, Prediction: {self._predicted_outcome}
                  Satisfaction: {self._predicted_outcome == _outcome}")

        """ Computing the next action to enact """
        # TODO: Implement the agent's decision mechanism
        self._action = 0
        # TODO: Implement the agent's anticipation mechanism
        self._predicted_outcome = 0
    return self._action
```

### Environment1 class

```
In [12]: class Environment1:
    """ In Environment 1, action 0 yields outcome 0, action 1 yields outcome 1 """
    def outcome(self, _action):
        # return int(input("entre 0 1 ou 2"))
        if _action == 0:
            return 0
```

```
    else:  
        return 1
```

## Environment2 class

```
In [13]: class Environment2:  
    """ In Environment 2, action 0 yields outcome 1, action 1 yields outcome 0 """  
    def outcome(self, _action):  
        if _action == 0:  
            return 1  
        else:  
            return 0
```

## Instantiate the agent

```
In [14]: a = Agent()
```

## Instantiate the environment

```
In [15]: e = Environment1()
```

## Test run the simulation

```
In [16]: outcome = 0  
for i in range(10):  
    action = a.action(outcome)  
    outcome = e.outcome(action)
```

```
Action: 0, Prediction: 0, Outcome: 0, Satisfaction: True  
Action: 0, Prediction: 0, Outcome: 0, Satisfaction: True
```

Observe that, on each interaction cycle, the agent correctly predicts the outcomes. The agent's satisfaction is True because its predictions are correct.

## PRELIMINARY EXERCISE

Run the agent in Environment2. Observe that its satisfaction becomes False. This agent is not satisfied in Environment2!

Now you see the goal of this assignment: design an agent that learns to be satisfied when it is run either in Environment1 or in Environment2.

# ASSIGNMENT

Implement Agent1 that:

- learns to predict the outcome of its actions
- chooses a different action when its predictions have been correct for 4 times in a row

The agent can choose two possible actions 0 or 1 , and can receive two possible outcomes: 0 or 1 .

It computes the prediction on the assumption that the same action always yields the same outcome in a given environment. You must thus implement a memory of the obtained outcomes for each action.

## Create your own agent by overriding the class Agent

Create an agent that learns to correctly predict the outcome of its actions in both Environment1 and Environment2.

You may add any class attribute or method you deem useful.

```
In [17]: class Agent1(Agent):  
    def __init__(self):  
        """Creating our learning agent that avoids boredom"""  
        super().__init__()  
        # Memory: stores the last observed outcome for each action  
        self.memory = {} # {action: outcome}  
        # Counter for consecutive correct predictions  
        self.correct_count = 0  
        # Boredom threshold  
        self.boredom_threshold = 4  
  
    def action(self, _outcome):  
        """Tracing the previous cycle"""  
        if self._action is not None:  
            # Check if prediction was correct  
            satisfied = (self._predicted_outcome == _outcome)  
  
            # Update correct prediction counter  
            if satisfied:  
                self.correct_count += 1  
            else:  
                self.correct_count = 0  
  
            # Check for boredom  
            bored = (self.correct_count >= self.boredom_threshold)  
  
            print(f"Action: {self._action}, Prediction: {self._predicted_outcome}  
                  f"Outcome: {_outcome}, Satisfaction: {satisfied}, Bored: {bore  
                  # Update memory with the observed outcome  
                  self.memory[self._action] = _outcome
```

```
"""Computing the next action to enact"""

# Implement the agent's decision mechanism
if self.correct_count >= self.boredom_threshold:
    # Switch action to avoid boredom
    self._action = 1 - self._action # Toggle between 0 and 1
    self.correct_count = 0 # Reset boredom counter
elif self._action is None:
    self._action = 0

# Implement the agent's anticipation mechanism
if self._action in self.memory:
    self._predicted_outcome = self.memory[self._action]
else:
    self._predicted_outcome = 0

return self._action
```

# Test your agent in Environment1

```
In [18]: a = Agent1()
          e = Environment1()
          outcome = 0
          for i in range(20):
              action = a.action(outcome)
              outcome = e.outcome(action)
```

# Test your agent in Environment2

```
In [19]: a = Agent1()
          e = Environment2()
          outcome = 0
          for i in range(20):
              action = a.action(outcome)
              outcome = e.outcome(action)
```

```
Action: 0, Prediction: 0, Outcome: 1, Satisfaction: False, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: True
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: False
Action: 1, Prediction: 0, Outcome: 0, Satisfaction: True, Bored: True
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Satisfaction: True, Bored: False
```

## Report

Explain what you programmed and what results you observed. Export this document as PDF including your code, the traces you obtained, and your explanations below (no more than a few paragraphs):

## Rapport — Agent 1

Nous avons implémenté un agent simple capable d'apprendre la relation entre une action et son outcome dans un environnement déterministe. Pour chaque action, l'agent mémorise le dernier outcome observé dans `self.memory` et s'en sert pour anticiper `self._predicted_outcome` l'issue du cycle suivant. En cas de bonne anticipation, il incrémente le compteur `self.correct_count`. Si ce compteur atteint 4, l'agent estime s'ennuyer et change d'action `self._action` pour explorer une autre possibilité.

Nous avons choisi une prédiction initiale fixée à 0 afin de rendre le comportement de départ plus prévisible, comme dans `world.py`. Une prédiction initiale aléatoire aurait aussi été envisageable pour favoriser une exploration dès le premier cycle.

## Environment 1

Dans Environment 1, l'agent exécute l'action 0 et anticipe correctement l'outcome 0 pendant 4 cycles consécutifs. Au 4<sup>e</sup> cycle, il détecte l'ennui ( `Bored = True` ), puis au 5<sup>e</sup> cycle, il change d'action, commet une erreur et ajuste immédiatement sa prédiction. À partir de là, le même schéma se répète à chaque série d'interactions : 4 cycles corrects, ennui au 4<sup>e</sup>, puis changement d'action au 5<sup>e</sup>, ce qui montre qu'il a appris et compris la structure de son environnement.

## Environment 2

Dans Environnement 2, l'agent commence par exécuter l'action `0` mais anticipe initialement un outcome `0`, ce qui provoque une erreur au 1<sup>er</sup> cycle `Satisfaction = False`. Dès le 2<sup>e</sup> cycle, il ajuste sa prédiction et atteint une satisfaction stable pendant 4 cycles, jusqu'à ce qu'il s'ennuie au 5<sup>e</sup> et change d'action au 6<sup>e</sup>. N'ayant encore rien mémorisé pour l'action `1`, il anticipe par défaut `0`, ce qui correspond ici à l'outcome réel `1 → 0`, et se retrouve satisfait immédiatement. Il reproduit ensuite le même schéma régulier : 4 cycles corrects, ennui au 4<sup>e</sup>, changement au 5<sup>e</sup>, montrant ainsi sa capacité d'adaptation à un environnement aux relations inversées.

En conclusion, l'agent 1 :

- apprend et prédit correctement les outcomes,
- détecte l'ennui après 4 succès,
- alterne entre les actions pour éviter de s'ennuyer, s'adapte aussi bien aux environnements 1 et 2.

# THE AGENT WHO THRIVED ON GOOD VIBES

## Learning objectives

Upon completing this lab, you will be able to implement agents driven by a type of intrinsic motivation called 'interactional motivation.' This refers to the drive to engage in sensorimotor interactions that have a positive valence while avoiding those that have a negative valence.

## Setup

### Define the Agent class

```
In [1]: class Agent:
    def __init__(self, _valences):
        """ Creating our agent """
        self._valences = _valences
        self._action = None
        self._predicted_outcome = None

    def action(self, _outcome):
        """ tracing the previous cycle """
        if self._action is not None:
            print(f"Action: {self._action}, Prediction: {self._predicted_outcome}
                  Prediction: {self._predicted_outcome == _outcome}, Valence:

        """ Computing the next action to enact """
        # TODO: Implement the agent's decision mechanism
        self._action = 0
        # TODO: Implement the agent's anticipation mechanism
        self._predicted_outcome = 0
    return self._action
```

### Environment1 class

```
In [2]: class Environment1:
    """ In Environment 1, action 0 yields outcome 0, action 1 yields outcome 1 """
    def outcome(self, _action):
        # return int(input("entre 0 1 ou 2"))
        if _action == 0:
            return 0
        else:
            return 1
```

## Environment2 class

```
In [3]: class Environment2:  
    """ In Environment 2, action 0 yields outcome 1, action 1 yields outcome 0 """  
    def outcome(self, _action):  
        if _action == 0:  
            return 1  
        else:  
            return 0
```

## Define the valence of interactions

```
In [4]: valences = [[-1, 1],  
                  [1, -1]]
```

The valence table specifies the valence of each interaction. An interaction is a tuple (action, outcome):

	<b>outcome 0</b>	<b>outcome 1</b>
action 0	-1	1
action 1	1	-1

## Instantiate the agent

```
In [5]: a = Agent(valences)
```

## Instantiate the environment

```
In [6]: e = Environment1()
```

Test run the simulation

```
In [7]: outcome = 0
for i in range(10):
    action = a.action(outcome)
    outcome = e.outcome(action)
```

Observe that, on each interaction cycle, the agent is mildly satisfied. On one hand, the agent made correct predictions, on the other hand, it experienced negative valence.

## PRELIMINARY EXERCISE

Execute the agent in Environment2. Observed that it obtains a positive valence.

Modify the valence table to give a positive valence when the agent selects action `0` and obtains outcome `0`. Observe that this agent obtains a positive valence in Environment1.

## ASSIGNMENT

Implement Agent2 that selects actions that, it predicts, will result in an interaction that have a positive valence.

Only when the agent gets bored does it select an action which it predicts to result in an interaction that have a negative valence.

In the trace, you should see that the agent learns to obtain a positive valence during several interaction cycles. When the agent gest bored, it occasionnaly selects an action that may result in a negative valence.

### Create Agent2 by overriding the class Agent

```
In [8]: class Agent2(Agent):
    def __init__(self, _valences):
        """Creating our hedonist agent"""
        super().__init__(_valences)
        # Memory: stores the last observed outcome for each action
        self.memory = {}
        # Counter for consecutive correct predictions
        self.correct_count = 0
        # Boredom threshold
        self.boredom_threshold = 4

    def action(self, _outcome):
        """Tracing the previous cycle"""
        if self._action is not None:
            # Update memory with the observed outcome
            self.memory[self._action] = _outcome

            # Check if prediction was correct
            satisfied = (self._predicted_outcome == _outcome)

            # Update correct prediction counter
            if satisfied:
                self.correct_count += 1
            else:
                self.correct_count = 0

            # Calculate valence
```

```

valence = self._valences[self._action][_outcome]

# Check for boredom
bored = (self.correct_count >= self.boredom_threshold)

print(f"Action: {self._action}, Prediction: {self._predicted_outcome}
      f"Outcome: {_outcome}, Prediction: {satisfied}, Valence: {vale

"""\Computing the next action to enact"""

# Implement the agent's decision mechanism
if self.correct_count >= self.boredom_threshold:
    # Bored: try a different action
    self._action = 1 - self._action
    self.correct_count = 0
else:
    # Not bored: choose the action with the best anticipated valence
best_action = None
best_valence = -float('inf')

for action in [0, 1]:
    if action in self.memory:
        predicted_outcome = self.memory[action]
        predicted_valence = self._valences[action][predicted_outcome]

        if predicted_valence > best_valence:
            best_valence = predicted_valence
            best_action = action

    if best_action is not None:
        self._action = best_action
    elif self._action is None:
        self._action = 0

# Implement the agent's anticipation mechanism
if self._action in self.memory:
    self._predicted_outcome = self.memory[self._action]
else:
    self._predicted_outcome = 0

return self._action

```

## Test your Agent2 in Environment1

```
In [14]: valences = [[-1, 1],
                  [1, -1]]

a = Agent2(valences)
e = Environment1()
outcome = 0
for i in range(20):
    action = a.action(outcome)
    outcome = e.outcome(action)
```

```
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: True
Action: 1, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: True
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: True
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: True
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
```

## Test your Agent2 in Environment2

```
In [10]: a = Agent2(valences)
```

```
e = Environment2()
outcome = 0
for i in range(20):
    action = a.action(outcome)
    outcome = e.outcome(action)
```

```
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 1, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1, Bored: False
Action: 0, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
```

## Test your agent with a different valence table

Note that, depending on the valence that you define, it may be impossible for the agent to obtain a positive valence in some environments.

```
In [16]: valences = [[-1, 1],
                  [-1, 1]]
# agent 2 environment 1
a = Agent2(valences)
e = Environment1()
outcome = 0
for i in range(20):
    action = a.action(outcome)
    outcome = e.outcome(action)
```

```
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: True
Action: 1, Prediction: 0, Outcome: 1, Prediction: False, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: True
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: -1, Bored: False
Action: 1, Prediction: 1, Outcome: 1, Prediction: True, Valence: 1, Bored: False
```

## Report

Explain what you programmed and what results you observed. Export this document as PDF including your code, the traces you obtained, and your explanations below (no more than a few paragraphs):

## Rapport — Agent 2

Pour cette partie, nous avons étendu le comportement de l'agent en intégrant la valence comme critère de décision. L'agent ne se contente plus d'anticiper les outcomes. Il choisit désormais l'action qui maximise la valence prédite à partir de la table `self._valences`. Lorsqu'il s'ennuie après 4 prédictions correctes consécutives, il sélectionne une autre action, même si celle-ci est associée à une valence négative.

## Environment 1

Dans Environment 1, les valences définies (`valences = [[-1, 1], [1, -1]]`) attribuent une valence négative à toutes les interactions possibles. Quelle que soit l'action effectuée (0 ou 1), l'outcome obtenu conduit systématiquement à une valence de -1. Ainsi, il est impossible pour l'agent d'obtenir une valence positive dans cet environnement avec cette configuration.

L'agent prédit correctement les outcomes dès le début, mais la valence reste négative.

Lors du choix d'action, lorsqu'il compare les valences anticipées, il rencontre l'action **0** en premier et conserve cette action puisqu'elle offre la même valence que l'action **1**. Le code ne prévoit pas de changer d'action en cas d'égalité, ce qui explique pourquoi il reste sur l'action **0** tant qu'il n'est pas ennuyé.

Il alterne ensuite ses actions uniquement en réponse à l'ennui détecté au 4<sup>e</sup> cycle (`Bored = True`), avant de revenir sur l'action **0** lors du prochain choix. Ce comportement engendre un schéma cyclique stable : 4 cycles corrects avec une valence constante de **-1**, ennui au 4<sup>e</sup>, changement d'action au 5<sup>e</sup>, puis retour sur l'action **0** et répétition du même pattern.

## Environment 2

Dans Environment 2, avec la table de valences `valences = [[-1, 1], [1, -1]]`, l'agent commence par exécuter l'action **0** avec une anticipation initiale incorrecte (`Prediction: 0`), ce qui entraîne une erreur dès le 1<sup>er</sup> cycle. Dès le 2<sup>e</sup> cycle, il ajuste sa prédiction à **1**, ce qui correspond à l'outcome réel (**0** → **1**) et lui permet d'obtenir une valence positive de **+1**.

L'action **1** offre également une valence de **+1** (**1** → **0**), mais l'agent privilégie l'action **0** car le mécanisme de décision explore les actions dans l'ordre et sélectionne la première qui présente la meilleure valence. Comme les deux actions ont la même valence positive, l'action **0** est systématiquement retenue. L'agent ne bascule donc sur l'action **1** que lorsqu'il détecte l'ennui au 4<sup>e</sup> cycle (`Bored = True`).

Après avoir changé d'action à cause de l'ennui, l'agent obtient une valence positive avec **1**, identique à celle de l'action **0**. Cependant, à cause de notre logique de code, au cycle suivant, il revient systématiquement sur l'action **0**. Le schéma devient ainsi régulier avec 4 cycles sur l'action **0**, ennui, passage à l'action **1**, puis retour à l'action **0**.

## Environment 1 avec valence différente `valences = [[-1, 1], [-1, 1]]`

Au départ l'agent ne sait pas que l'action **1** peut donner une valence positive avec l'outcome **1**. N'ayant encore rien mémorisé pour l'action **1**, il reste sur l'action **0**, ce qui lui donne une valence négative de **-1** pendant 4 cycles. Lorsque l'ennui apparaît au 4<sup>e</sup> cycle, il essaie l'action **1**, obtient une valence positive de **+1** et mémorise cette association favorable.

Dès que cette information est en mémoire, le mécanisme de décision privilégie **1** (car elle maximise la valence anticipée) et l'agent y reste jusqu'au prochain ennui. Lorsqu'il s'ennuie à nouveau, il fait un bref passage par l'action **0** (valence **-1**) puis revient rapidement sur l'action **1** qui reste la meilleure option.

Schéma observé 4 cycles sur **0** (valence **-1**), puis ennui, puis passage à **1** (valence **+1**). Après ce changement, l'agent maintient ses cycles sur **1** jusqu'au prochain ennui, fait un bref détour par **0**, puis revient rapidement sur **1** pour reprendre un cycle stable

avec la valence positive.

## Conclusion 1

Nous avons réussi à mettre en place tout ce qui était demandé dans le TP. L'agent apprend à prédire les outcomes, cherche les interactions à valence positive et change d'action lorsqu'il s'ennuie.

On peut aussi rendre l'agent plus flexible. Par exemple, il pourrait choisir au hasard entre les actions qui ont la même valence max pour éviter de faire toujours le même choix.

Ensuite, on peut ajuster son comportement face à l'ennui. Il pourrait s'ennuyer plus vite au début pour explorer rapidement et découvrir les meilleures valences, puis réduire son ennui une fois une bonne interaction trouvée, surtout dans des environnements avec plusieurs actions possibles.

# THE AGENT WHO TAMED THE TURTLE

## Learning objectives

Upon completing this lab, you will be able to assign appropriate valences to interactions, enabling a developmental agent to exhibit exploratory behavior in a simulated environment.

## Setup

### Import the turtle environment

```
In [66]: !pip3 install ColabTurtle  
from ColabTurtle.Turtle import *
```

Requirement already satisfied: ColabTurtle in c:\users\youssef\miniconda3\envs\developmental-ai-lab\lib\site-packages (2.1.0)

### Define the Agent class

```
In [67]: class Agent:  
    def __init__(self, _valences):  
        """ Creating our agent """  
        self._valences = _valences  
        self._action = None  
        self._predicted_outcome = None  
  
    def action(self, _outcome):  
        """ tracing the previous cycle """  
        if self._action is not None:  
            print(f"Action: {self._action}, Prediction: {self._predicted_outcome}  
                  Prediction: {self._predicted_outcome == _outcome}, Valence:  
  
        """ Computing the next action to enact """  
        # TODO: Implement the agent's decision mechanism  
        self._action = 0  
        # TODO: Implement the agent's anticipation mechanism  
        self._predicted_outcome = 0  
        return self._action
```

### Define the tutele environment class

You don't need to worry about the code of the ColabTurtleEnvironment below.

Just know that this environment:

- interprets the agent's actions as follows 0 : move forward, 1 : turn left, 2 : turn right.
- returns outcome 1 when the turtle bumps into the border of the window, and 0 otherwise.

```
In [68]: # @title Initialize the turtle environment

BORDER_WIDTH = 20

class ColabTurtleEnvironment:

    def __init__(self):
        """ Creating the Turtle window """
        bgcolor("lightGray")
        penup()
        goto(window_width() / 2, window_height()/2)
        face(0)
        pendown()
        color("green")

    def outcome(self, action):
        """ Enacting an action and returning the outcome """
        _outcome = 0
        for i in range(10):
            # _outcome = 0
            if action == 0:
                # move forward
                forward(10)
            elif action == 1:
                # rotate left
                left(4)
                forward(2)
            elif action == 2:
                # rotate right
                right(4)
                forward(2)

        # Bump on screen edge and return outcome 1
        if xcor() < BORDER_WIDTH:
            goto(BORDER_WIDTH, ycor())
            _outcome = 1
        if xcor() > window_width() - BORDER_WIDTH:
            goto(window_width() - BORDER_WIDTH, ycor())
            _outcome = 1
        if ycor() < BORDER_WIDTH:
            goto(xcor(), BORDER_WIDTH)
            _outcome = 1
        if ycor() > window_height() - BORDER_WIDTH:
            goto(xcor(), window_height() - BORDER_WIDTH)
            _outcome = 1

        # Change color
        if _outcome == 0:
            color("green")
        else:
            # Finis l'interaction
            color("red")
```

```

# if action == 0:
#     break
if action == 1:
    for j in range(10):
        left(4)
elif action == 2:
    for j in range(10):
        right(4)
break

return _outcome

```

## Define the valence of interactions

```
In [69]: valences = [[1, -1],
                  [-1, -1],
                  [-1, -1]]
```

The valence table specifies the valence of each interaction. An interaction is a tuple (action, outcome):

	0 Not bump	1 Bump
0 Forward	1	-1
1 Left	-1	-1
2 Right	-1	-1

## Instantiate the agent

```
In [70]: a = Agent(valences)
```

## Run the simulation

```
In [71]: # @title Run the simulation

initializeTurtle()

# Parameterize the rendering
bgcolor("lightGray")
penup()
goto(window_width() / 2, window_height()/2)
face(0)
pendown()
color("green")
speed(10)

e = ColabTurtleEnvironment()

outcome = 0
for i in range(10):
    action = a.action(outcome)
    outcome = e.outcome(action)
```



```
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1  
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1  
Action: 0, Prediction: 0, Outcome: 0, Prediction: True, Valence: 1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1  
Action: 0, Prediction: 0, Outcome: 1, Prediction: False, Valence: -1
```

Observe the turtle moving in a straight line until it bumps into the border of the window

## PRELIMINARY EXERCISE

Copy Agent2 that you designed in your previous assignment to this notebook.

Observe how your Agent2 behaves in this environment

## ASSIGNMENT

Implement Agent3 by modifying your previous Agent2 such that it can select 3 possible actions: 0 , 1 , or 2 .

Choose the valences of interactions so that the agent does not remain stuck in a corner of the environment.

Create Agent3 by overriding the class Agent or your previous class Agent2

In [79]: `import random`

```

class Agent3(Agent):
    def __init__(self, _valences):
        """Creating our turtle-taming agent"""
        super().__init__(_valences)
        # Mémoire pour stocker le dernier outcome observé pour chaque action
        self.memory = {}
        # Compteur de prédictions correctes consécutives
        self.correct_count = 0
        # Seuil d'ennui (fixe ici, mais tu peux le rendre progressif si besoin)
        self.boredom_threshold = 4
        # Pour mémoriser l'action qui a provoqué l'ennui
        self.last_bored_action = None

    def action(self, _outcome):
        """Tracing the previous cycle"""
        if self._action is not None:
            # Met à jour la mémoire avec le dernier outcome observé
            self.memory[self._action] = _outcome

            # Vérifie si la prédiction était correcte
            satisfied = (self._predicted_outcome == _outcome)

            # Met à jour le compteur de prédictions correctes
            if satisfied:
                self.correct_count += 1
            else:
                self.correct_count = 0

            # Calcule la valence
            valence = self._valences[self._action][_outcome]

            # Vérifie si l'agent s'ennuie
            bored = (self.correct_count >= self.boredom_threshold)

            print(f"Action: {self._action}, Prediction: {self._predicted_outcome}
                  f"Outcome: {_outcome}, Prediction: {satisfied}, "
                  f"Valence: {valence}, Bored: {bored}")
        else:
            bored = False

        """Decision mechanism"""
        if bored:
            # Sauvegarde de l'action ennuyeuse
            self.last_bored_action = self._action

            # Sélectionne aléatoirement une autre action que celle ennuyeuse
            possible_actions = [0, 1, 2]
            if self.last_bored_action in possible_actions:
                possible_actions.remove(self.last_bored_action)
            self._action = random.choice(possible_actions)
            self.correct_count = 0

        else:
            # Sélection des actions avec la meilleure valence anticipée
            best_valence = -float('inf')
            best_actions = []

            for action in [0, 1, 2]:
                if action in self.memory:
                    predicted_outcome = self.memory[action]

```

```

predicted_valence = self._valences[action][predicted_outcome]

    if predicted_valence > best_valence:
        best_valence = predicted_valence
        best_actions = [action]
    elif predicted_valence == best_valence:
        best_actions.append(action)

# Tirage aléatoire équitable si plusieurs actions ont la même valence
if best_actions:
    self._action = random.choice(best_actions)
elif self._action is None:
    self._action = 0

"""Anticipation mechanism"""
if self._action in self.memory:
    self._predicted_outcome = self.memory[self._action]
else:
    self._predicted_outcome = 0

return self._action

```

## Choose the valence table

Replace the `valences` table by your choice in the code below

```
In [80]: valences = [[1, -1],
                  [-1, 1],
                  [-1, 1]]
```

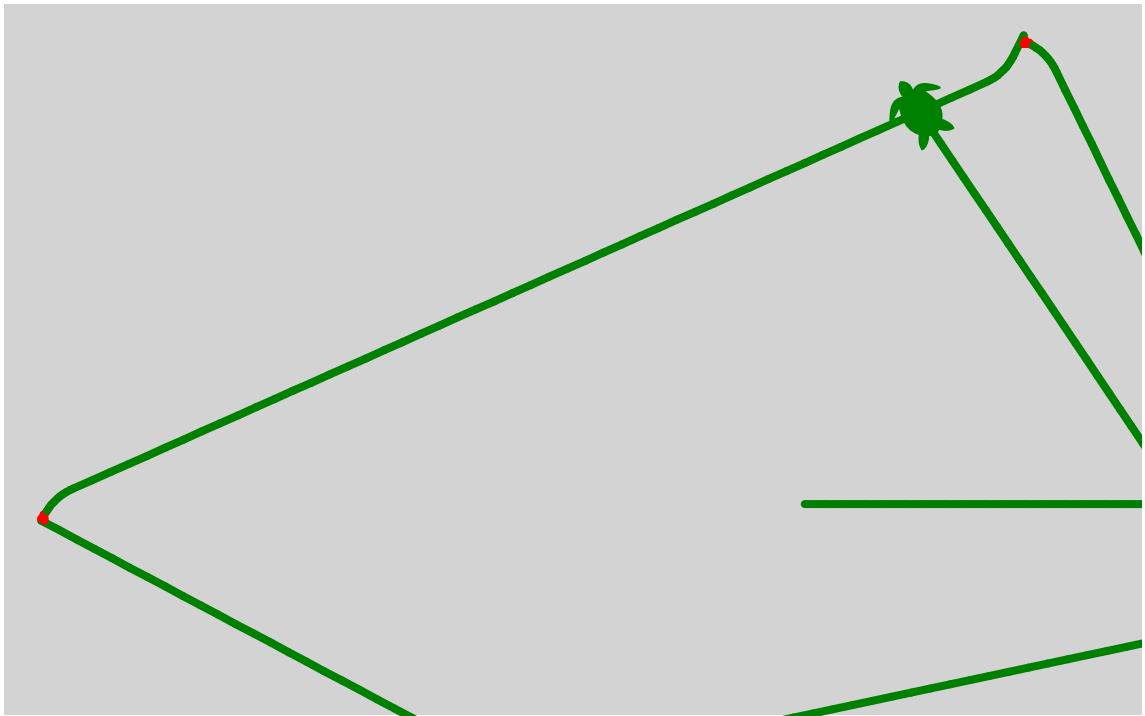
## Test your agent in the TurtleEnvironment

```
In [82]: initializeTurtle()

# Parameterize the rendering
bgcolor("lightGray")
penup()
goto(window_width() / 2, window_height()/2)
face(0)
pendown()
color("green")
speed(10)

a = Agent3(valences)
e = ColabTurtleEnvironment()

outcome = 0
for i in range(50):
    action = a.action(outcome)
    outcome = e.outcome(action)
```





## Improve your agent's code

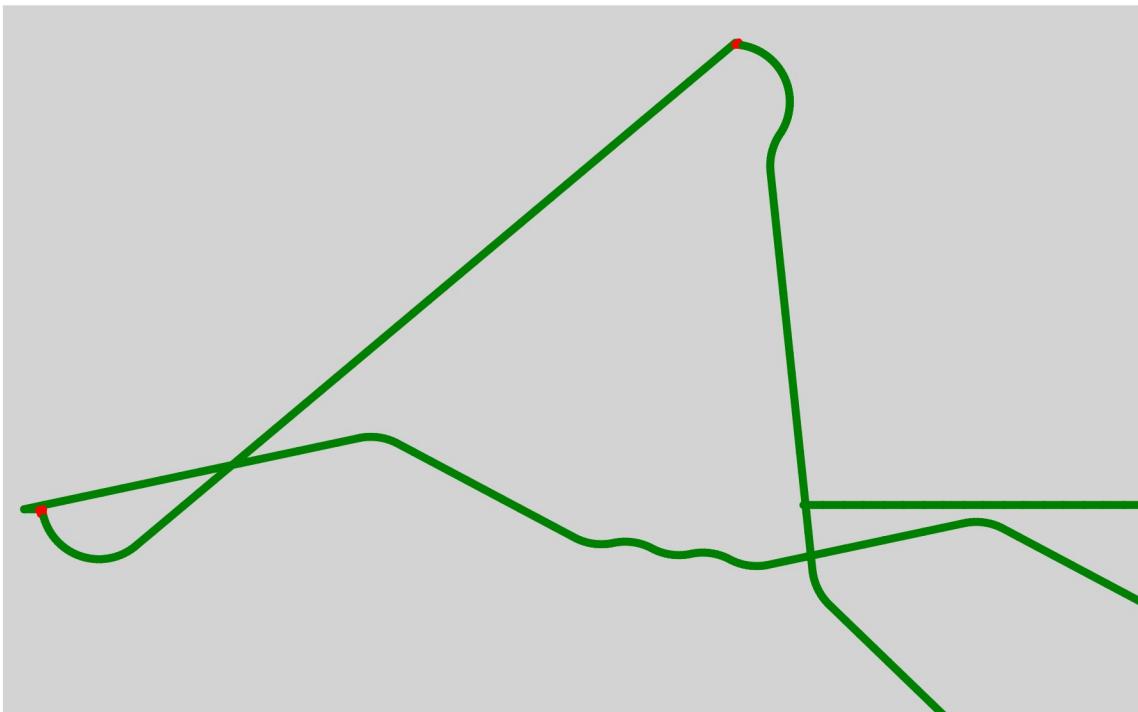
If your agent gets stuck against a border or in a corner, modify the valences or the code. Try different ways to handle boredom or to select random actions. In the next lab, you will see how to design an agent that can adapt to the context.

```
In [87]: valences = [
    [-1, -1],
    [-1, 1],
    [-1, 1]
]

initializeTurtle()
# Parameterize the rendering
bgcolor("lightGray")
penup()
goto(window_width() / 2, window_height()/2)
face(0)
pendown()
color("green")
speed(10)

a = Agent3(valences)
e = ColabTurtleEnvironment()

outcome = 0
for i in range(50):
    action = a.action(outcome)
    outcome = e.outcome(action)
```





# Report

Explain what you programmed and what results you observed. Export this document as PDF including your code, the traces you obtained, and your explanations below (no more than a few paragraphs):

## Rapport — Agent 3

### Exercice préliminaire : Agent2 dans TurtlePy

Avec l'agent2, la tortue ne peut qu'avancer ou tourner à gauche. Elle ne peut jamais tourner à droite, donc elle a moins de possibilités de mouvement. Quand elle touche un mur, elle tourne toujours dans le même sens et à plus de chance de rester bloquée dans un coin.

### Objectif

L'objectif principal de cette partie était d'améliorer Agent2 pour l'adapter à l'environnement TurtlePy afin qu'il évite au maximum de rester bloqué contre les murs.

Pour cela, nous avons implémenté Agent3 en étendant Agent2 afin de gérer trois actions au lieu de deux :

- Action 0 : Avancer
- Action 1 : Tourner à gauche
- Action 2 : Tourner à droite

L'agent utilise les mêmes mécanismes qu'Agent2 :

- Mémoire des outcomes,
- Anticipation à partir de cette mémoire,
- Décision basée sur la valence (choisit l'action à valence anticipée maximale),
- Gestion de l'ennui (changement d'action après 4 prédictions correctes consécutives).

### Améliorations et tentatives d'optimisation dans le code

#### Tirage aléatoire en cas d'égalité de valence

Si plusieurs actions ont la même valence maximale, l'agent effectue désormais un tirage aléatoire équitable parmi elles. Cette amélioration empêche l'agent de sélectionner systématiquement la première action rencontrée et de rester bloqué dessus lorsqu'elle a la même valence que d'autres. Cela rend son comportement plus flexible et lui permet d'explorer davantage l'environnement.

## Test de variation de l'ennui avec epsilon

Nous avons également testé une variation de l'ennui avec une variable epsilon, afin de rendre le comportement de l'agent plus aléatoire et dynamique. Cependant, les résultats n'étaient pas concluants et cette approche a été abandonnée dans la version finale de l'Agent3.

## Analyse des comportements observés

### Cas 1 — (`valences = [[ 1, -1],[-1, 1], [-1, 1]]`)

Dans cette configuration, les actions `1` et `2` (tourner à gauche ou à droite) offrent la même valence positive `+1` lorsqu'elles sont utilisées après une collision. Cela donne à l'agent la possibilité de tourner dès qu'il tape un mur.

Au début, l'agent n'a aucune information sur les actions disponibles. Il commence naturellement par l'action `0` (avancer), qui lui donne une valence positive tant qu'il ne rencontre pas d'obstacle. Lorsque la tortue se cogne contre le bord, la valence de l'action `0` devient négative, mais comme les actions `1` et `2` n'ont pas encore été essayées, l'agent continue à avancer. Il reste donc bloqué contre le mur jusqu'à atteindre le seuil d'ennui.

Une fois ce seuil atteint, l'agent choisit aléatoirement entre les actions `1` et `2` dans notre cas `1`. Cela lui permet de découvrir une nouvelle action de rotation et de se dégager de l'obstacle. Par la suite, il alterne entre avancer (`0`) et tourner (`1`) selon la situation rencontrée. Grâce au même mécanisme d'ennui, il finit aussi par tester la deuxième action de rotation.

Le fait que les actions `1` et `2` aient la même valence maximale permet au tirage aléatoire d'éviter que l'agent reste fixé sur une seule direction lorsque qu'il rentre en collision avec un mur. Ce comportement rend la navigation moins rigide et améliore sa capacité à éviter de rester bloqué contre les murs.

### Cas 2 — (`valences = [[ -1, -1],[-1, 1], [-1, 1]]`)

Dans ce scénario, l'action `0` (avancer) a une valence négative dans toutes les situations, que ce soit en avançant librement ou en heurtant un mur. En revanche, les actions `1` et `2` (tourner à gauche et à droite) ont une valence positive lorsqu'elles sont utilisées après une collision.

Au début, l'agent ne connaît aucune action et commence donc naturellement par l'action `0`. Comme cette action est toujours associée à une valence négative, il accumule rapidement des interactions négatives. Malgré cela, il persiste sur cette action puisqu'il n'a encore rien appris sur les autres.

Lorsque le seuil d'ennui est atteint, il explore aléatoirement une autre action, soit `1` soit

`2` dans notre cas `2`. C'est à ce moment qu'il découvre une valence positive lorsqu'il tourne après une collision, ce qui en fait une meilleure option que l'action `0` dans ces situations. Par la suite, grâce à de nouveaux épisodes d'ennui, il finit par découvrir la deuxième action de rotation restante `1`.

Une fois qu'il a mémorisé les trois actions, son comportement devient plus varié. En dehors des collisions, toutes les actions ont une valence égale à `-1`, ce qui entraîne un tirage aléatoire entre elles. Cela permet d'éviter un comportement déterministe et rend sa trajectoire plus vivante et moins linéaire.

De plus, grâce aux valences positives des actions gauche et droite lors des collisions, l'agent est capable de se débloquer efficacement lorsqu'il rencontre un mur. Il peut ainsi continuer à explorer l'environnement tout en conservant des trajectoires dynamiques et variées.

## Conclusion

En résumé, l'agent commence avec une stratégie simple consistant à avancer. Grâce au mécanisme d'ennui et au tirage aléatoire, il découvre progressivement les actions de rotation, ce qui lui permet d'adopter un comportement plus flexible et mieux adapté à l'environnement TurtlePy.

La première configuration de valences (`[[ 1, -1], [ -1, 1], [ -1, 1]]`) est efficace. Elle encourage l'agent à avancer lorsqu'il n'y a pas d'obstacle et à changer de direction lorsqu'il rencontre un mur. Cela donne des trajectoires simples et efficaces.

La deuxième configuration (`[[ -1, -1], [ -1, 1], [ -1, 1]]`) est encore plus intéressante. Elle permet à l'agent d'avoir des trajectoires plus naturelles et moins robotiques, en alternant plus souvent entre les différentes actions. Elle permet aussi la capacité de l'agent à se débloquer contre les murs tout en explorant l'environnement de manière plus variée.