

# Puissance 4

Christophe Viroulaud

Première - NSI

**Lang 08**

Identifier les  
besoins

Modéliser

Implémenter



FIGURE 1 – Le *Puissance 4* est un jeu de stratégie en duel.

Identifier les  
besoins

Modéliser

Implémenter

Comment construire un projet ?

Identifier les  
besoins

Modéliser

Implémenter

1. Identifier les besoins

2. Modéliser

3. Implémenter

Identifier les  
besoins

Modéliser

Implémenter

## À retenir

Il s'agit de définir les **spécifications** du jeu.

- ▶ une grille de 7 colonnes et 6 lignes,
- ▶ 2 joueurs en alternance (rouge et jaune),
- ▶ gagnant : 4 pions horizontaux ou verticaux.

## Remarque

Dans cette activité on construira une version simplifiée du jeu : on ne regardera pas les pions alignés en diagonal.

Identifier les  
besoins

Modéliser

Implémenter

1. Identifier les besoins

2. **Modéliser**

3. Implémenter

# Modéliser - conception générale

Identifier les  
besoins

Modéliser

Implémenter

## À retenir

Il s'agit de définir un **algorithme général** du jeu.

**Activité 1** : Construire un déroulé du jeu.



# Correction

Identifier les  
besoins

Modéliser

Implémenter

Initialiser la grille

Choisir un joueur de départ.

Tant qu'il n'y a pas de gagnant :

- ▶ Demander la colonne choisie.
- ▶ Vérifier que la colonne n'est pas pleine.
- ▶ Placer le jeton en le *laissant tomber* dans la colonne.
- ▶ Vérifier si le placement est gagnant :
  - ▶ si oui : partie terminée,
  - ▶ si non : changement de joueur.

Identifier les  
besoins

Modéliser

Implémenter

## À retenir

Il s'agit de donner les **signatures** des fonctions nécessaires.

## Initialiser la grille

Choisir un joueur de départ.

Tant qu'il n'y a pas de gagnant :

- ▶ **Demander** la colonne choisie.
- ▶ **Vérifier** que la colonne n'est pas pleine.
- ▶ **Placer** le jeton en le *laissant tomber* dans la colonne.
- ▶ **Vérifier** si le placement est gagnant :
  - ▶ si oui : partie terminée,
  - ▶ si non : changement de joueur.

**Activité 2 :** Donner une signature pour chaque étape de l'algorithme.

Identifier les  
besoins

Modéliser

Implémenter

Identifier les  
besoins

Modéliser

Implémenter

► `initialiser_grille() -> list`

Identifier les  
besoins

Modéliser

Implémenter

- ▶ `initialiser_grille()` -> list
- ▶ `choisir_colonne()` → int

# Correction

Identifier les  
besoins

Modéliser

Implémenter

- ▶ `initialiser_grille()` -> list
- ▶ `choisir_colonne()` → int
- ▶ `est_replie(grille: list, colonne: int) → bool`

Identifier les  
besoins

Modéliser

Implémenter

- ▶ `initialiser_grille()` -> list
- ▶ `choisir_colonne()` → int
- ▶ `est_replie(grille: list, colonne: int)` → bool
- ▶ `placer_jeton(grille: list, colonne: int, joueur)` → int

Identifier les  
besoins

Modéliser

Implémenter

- ▶ `initialiser_grille()` → `list`
- ▶ `choisir_colonne()` → `int`
- ▶ `est_replie(grille: list, colonne: int)` → `bool`
- ▶ `placer_jeton(grille: list, colonne: int, joueur)` → `int`
- ▶ `verif_gagnant(grille: list, joueur: int, ligne: int, colonne: int)` → `bool`



## Remarque

Il sera peut-être nécessaire d'écrire d'autres fonctions *internes* pour exécuter certaines tâches, rendre le code plus lisible...

Identifier les  
besoins

Modéliser

Implémenter

1. Identifier les besoins

2. Modéliser

3. Implémenter

Identifier les  
besoins

Modéliser

Implémenter

## À retenir

Il s'agit de **transformer en code informatique** l'algorithme modélisé.

### Activité 3 :

1. Télécharger et extraire le dossier compressé `puissance4-annexe.zip` sur le site <https://cviroulaud.github.io>
2. Ouvrir le fichier `puissance4.py`

Identifier les  
besoins

Modéliser

Implémenter

Le programme principal implémente l'algorithme général.

```
1 grille = initialiser_grille()  
2 joueur = ROUGE
```

### Code 1 – Initialisation

## Remarque

La couleur (ROUGE) du joueur est stockée dans une **constante**.

```
1 remplie = True
2 while remplie:
3     colonne = choisir_colonne()
4     remplie = est_remplie(grille, colonne)
```

Code 2 – Demander la colonne et vérifier

## Remarque

Il faut initialiser la variable `remplie`.

```
1 ligne = placer_jeton(grille, colonne, joueur)
```

Code 3 – Placer le jeton

## Remarque

On récupère la valeur de la `ligne`.



```
1  if verif_gagnant(grille, joueur, ligne, colonne):  
2      gagnant = True  
3  else:  
4      # au tour de l'autre joueur  
5      joueur = changer_joueur(joueur)
```

Code 4 – Vérifier le gagnant

## Remarques

- ▶ Le code est découpé en fichiers puis en fonctions.
- ▶ Le programme principal est simplifié au maximum.
- ▶ La partie *graphique* est pour l'instant hors programme.

## Activité 4 :

1. Ouvrir le fichier `constantes.py`. Il contient des variables utilisables dans tout le programme. Elles ne doivent pas être modifiées.
2. Ouvrir le fichier `fonctions_placement.py`
3. Compléter la fonction `initialiser_grille` en construisant la grille par compréhension.
4. Compléter la fonction `est_remplie` qui vérifie si la colonne est remplie.

Identifier les  
besoins

Modéliser

Implémenter

```
1 def initialiser_grille() -> list:
2     """
3     construire la grille du jeu
4
5     Returns:
6         list: un tableau de HAUTEUR lignes et
7             LARGEUR colonnes
8     """
9     return [[VIDE for i in range(LARGEUR)] for j in
10             range(HAUTEUR)]
```

Code 5 – Initialiser

Identifier les  
besoins

Modéliser

Implémenter

```
1 def est_remplie(grille: list, colonne: int) -> bool:
2     """
3     vérifie si la colonne est remplie jusqu'en haut
4
5     Args:
6         grille (list): le jeu
7         colonne (int): la colonne
8
9     Returns:
10        bool: True si la colonne est remplie
11    """
12    # il suffit de vérifier si l'emplacement le plus
13    haut est vide
14    return not(grille[0][colonne] == VIDE)
```

Code 6 – Colonne remplie ?

## Activité 5 :

1. Pour placer le jeton on écrit une fonction intermédiaire : `tomber_ligne(grille: list, colonne: int) → int`. Elle renvoie la position du jeton qui est tombé.
2. En utilisant la fonction précédente, compléter la fonction `placer_jeton`

```
1 def tomber_ligne(grille: list, colonne: int) -> int:
2     ligne = 0
3     while ligne < HAUTEUR and grille[ligne][colonne]
      == VIDE:
4         # on descend tant qu'on n'est pas en bas ou
      sur une case remplie
5         ligne = ligne + 1
6
7     # renvoie la dernière place vide
8     return ligne-1
```

Code 7 – Trouve la ligne d'arrivée

```
1 def placer_jeton(grille: list, colonne: int, joueur) -> int:  
2     ligne = tomber_ligne(grille, colonne)  
3     grille[ligne][colonne] = joueur  
4     return ligne
```

### Code 8 – Place le jeton



## Activité 6 : Étude du reste du code :

1. Comment fonctionne la fonction `verif_gagnant` ?
2. Dans la fonction `verif_verticale`, quelles sont les conditions pour que la boucle `while` soit exécutée ?
3. Que faut-il ajouter pour vérifier les diagonales ?

```
1  if verif_verticale(grille, joueur, ligne, colonne) or \  
2      verif_horizontale_droite(grille, joueur, ligne,  
    colonne) or \  
3      verif_horizontale_gauche(grille, joueur, ligne,  
    colonne):
```

## Code 9 – Gagnant ?

Pour gagner il suffit (or) qu'une des conditions soient vérifiées.

## Remarque

Pour vérifier si la partie est gagnée il suffit de regarder *vers le bas* de la grille.

```
1 while ligne < HAUTEUR and  
2     grille[ligne][colonne] == joueur and  
3     compteur < 4:
```

- ▶ on ne sort pas de la grille,
- ▶ les jetons sont de la même couleur,
- ▶ on n'a pas encore 4 jetons de même couleur.

Pour améliorer le jeu il faut :

- ▶ créer les fonctions `verif_diagonale`,
- ▶ modifier la condition de la fonction `verif_gagnant`.

```
1  if verif_verticale(grille, joueur, ligne, colonne) or \  
2      verif_horizontale_droite(grille, joueur, ligne,  
    colonne) or \  
3      verif_horizontale_gauche(grille, joueur, ligne,  
    colonne) or \  
4      verif_diagonale(grille, joueur, ligne, colonne):
```