Programmation avec PHP : remise dans le bain

Licence Informatique 3ème année

Alexandre Niveau — Jean-Marc Lecarpentier

Programmation avec PHP : remise dans le bain

Notes de cours

- **Introduction à PHP**
  - Syntaxe générale : variables, structures de contrôle, fonctions…
  - Tableaux associatifs
- **Introduction à PHP : compléments**
  - Divers détails spécifiques au langage
  - Lecture d'un fichier
- **Programmation objet avec PHP**

Travail personnel

Objectifs

Dans ce TP on revoit les bases du langage PHP, en l'utilisant depuis un terminal.

Précision importante : Les ex-L2 ayant déjà eu un cours d'introduction à PHP, les bases de ce langage sont considérées comme des **pré-requis** de ce cours. Cependant de nombreux (?) étudiant·es n'arrivent en L3 que maintenant, et ne connaissent pas forcément PHP. C'est principalement à ces étudiant·es que les deux premiers exercices de ce TP sont destinés, c'est pourquoi ils sont indiqués comme étant optionnels. Le troisième exercice n'est pas optionnel en revanche, autant pour les personnes débutantes que confirmées !

Exercice 1 (optionnel) — Boucles, tableaux et fonctions

en PHP

> Une variante de cet exercice a été donnée en L2. Il est optionnel, en particulier pour les ex-L2. Traitez-le si vous n'avez jamais fait de PHP ou si vous voulez vous rafraîchir la mémoire.

Cet exercice a pour objectif de vous faire manipuler la syntaxe de base de PHP en essayant parfois de vous faire tomber dans des pièges classiques.

❦ Échauffement

1. Créer un script PHP `test.php` qui affiche tous les nombres entiers de 1234 à 5678, avec un saut de ligne après chaque nombre.
2. Dans un script PHP, mettre dans une variable `$tab` un tableau (*array*) vide, et le remplir avec 100 valeurs en utilisant une boucle `for`, de façon à ce que la case d'indice *i* contienne la valeur 3×*i*+2.
3. Afficher le contenu de `$tab` en utilisant une boucle `foreach`.
4. Dans un script PHP, créer une fonction prenant en paramètre un tableau de nombres et renvoyant la moyenne arithmétique de ces nombres. Tester par exemple sur le tableau `$tab`.

❦ Tableaux associatifs

1. Dans un script PHP, créer un tableau associatif `$persos` indiquant la taille de quatre ou cinq personnages de fiction (les clefs seront les noms des personnages, et les valeurs des flottants). NB : ne perdez pas de temps à chercher des infos, ça n'a aucun intérêt… Vous pouvez même inventer des personnages…
2. En utilisant une boucle `foreach`, afficher la taille de chaque personnage sous la forme « *Personnage* mesure *x* cm. »
3. Écrire une fonction « minimum() » prenant en paramètre un tableau, qui cherche la plus petite valeur du tableau et la renvoie. Tester sur votre tableau `$persos`.
4. Ajouter un *paramètre optionnel* booléen `$return_key` à la fonction minimum, qui prend par défaut la valeur `false` (comme le veut l'usage pour les paramètres booléens). S'il est mis à `true`, on renvoie *la clef* correspondant à la plus petite valeur (et non la valeur elle-même). Tester sur votre tableau `$persos` : cela doit renvoyer le nom du personnage le plus petit.
5. Écrire une fonction « min_and_max() » qui renvoie cette fois un tableau de taille 2, contenant la valeur la plus petite et la valeur la plus grande du tableau donné en paramètre. Tester sur `$persos`.

❦ Fonctions de tri de tableaux

1. Afficher le tableau avec chacune des trois fonctions d'affichage debug : `print_r`, `var_dump`, `var_export`. Choisissez-en une pour la suite.
2. Appeler les fonctions suivantes sur le tableau, et regarder l'effet grâce à une des fonctions d'affichage :
   - asort() et son inverse arsort()
   - ksort() et son inverse krsort()
   - sort() et son inverse rsort()
   - shuffle()

   Si vous êtes perdu·e : plus d'infos sur les fonctions de tri de tableaux

❦ Recherche d'une valeur dans un tableau

La fonction `array_search($val, $tab)` cherche si le tableau $tab contient la valeur $val, et renvoie la première clef correspondante si c'est le cas, et `false` sinon.

Regardez le script suivant et **essayez de deviner** ce qui va s'afficher.

```php
<?php
function contient_toto($tab) {
    if (array_search('toto', $tab) == false) {
        echo "Le tableau ne contient pas toto\n";
    } else {
        echo "Le tableau contient toto !!\n";
    }
}

$x = [ 'titi', 'toto', 'tutu' ];
contient_toto($x);
$y = [ 'tutu', 'titi', 'tete' ];
contient_toto($y);
$z = [ 'toto', 'titi', 'tutu' ];
contient_toto($z);
$t = [ 'titi', 'tutu', 0  ];
contient_toto($t);
```

Recopiez-le ensuite dans un script, et exécutez-le : aviez-vous vu juste ? Si non, essayez de comprendre ce qui se passe. La page du manuel, en lien ci-dessus, peut être utile !

Exercice 2 (optionnel) — Chaînes de caractères en PHP

Une variante de cet exercice a été donnée en L2. Il est optionnel, en

particulier pour les ex-L2. Traitez-le si vous n'avez jamais fait de PHP ou si vous voulez vous rafraîchir la mémoire.

La fonction `readline` est l'équivalent de `input` en Python : elle attend que l'utilisateur/trice tape une ligne de texte dans le terminal, et retourne le résultat sous forme de chaîne de caractères.

1. Écrire un script PHP qui demande le nom et le prénom de l'utilisateur/trice, et lui dit ensuite une phrase de bienvenue personnalisée.
2. Le script doit ensuite afficher le nombre de caractères du prénom. Vérifiez bien que ça fonctionne même pour les prénoms comportant des caractères spéciaux, ou même des émoticônes, comme 🐳, qui doit être considéré comme un seul caractère.
3. Ajouter le code suivant à la fin de votre script (en adaptant si besoin les noms des variables) :

   ```
   if ($prenom + $nom == 'JeanValjean') {
       echo "Surprise !\n";
   }
   ```

   La surprise s'affiche-t-elle si vous dites que vous vous appelez Jean Valjean, ou l'inverse ? Comprenez-vous ce qui se passe ? Si vous ne comprenez pas, essayez de débugger…
4. Trouver la fonction PHP permettant de renvoyer l'indice du premier caractère « a » dans une chaîne. Compléter le script pour qu'il indique quelque chose comme « Vous avez un a en position 12 dans votre prénom » ou « Vous n'avez pas de a dans votre prénom », en fonction des cas. Vérifiez que ça marche pour les prénoms « jean », « albert » et « béatrice », en minuscules. **Optionnel** : faire en sorte que ça marche aussi correctement si le prénom commence par une majuscule.
5. La fonction `explode` découpe une chaîne de caractères selon un délimiteur donné et renvoie les morceaux dans un tableau. Demander à l'utilisateur/trice de rentrer une phrase, et en utilisant cette fonction, affichez le troisième mot de la phrase.
6. La fonction `implode` fait l'inverse de la précédente : elle s'applique à un tableau contenant des chaînes, et renvoie le résultat de la concaténation de ces chaînes, séparées par un séparateur donné en paramètre. L'équivalent en Python est la méthode `str.join`.

   En utilisant notamment cette fonction, affichez la phrase rentrée par l'utilisateur/trice, mais avec les mots rangés dans l'ordre alphabétique.

Exercice 3 — Rule 110, un automate cellulaire

élémentaire

> L'énoncé de cet exercice est long, mais c'est parce qu'il est très guidé ! Vous devez aller au moins jusqu'au prochain encadré (avant « Architecture pour les règles d'évolution »).

Dans cet exercice on va programmer l'*automate cellulaire élémentaire* appelé « rule 110 » (un des systèmes les plus simples qui soit *Turing-complet*, c'est-à-dire capable de simuler n'importe quel calcul).

❦ Contexte

Automate cellulaire élémentaire

On peut voir un automate cellulaire élémentaire comme la simulation d'un monde très simple. Ce monde est constitué de *cellules*, qui n'ont que deux états possibles, blanc ou noir (ou morte/vivante, ou vide/pleine, ou fausse/vraie… comme vous voulez). Elles sont organisées sur une ligne (le monde n'a donc qu'une seule dimension). En général on considère que les deux extrémités de la ligne se rejoignent — il s'agit donc plutôt d'un cercle, mais ce n'est pas très important ici.

À chaque pas de temps (ou à chaque « génération »), les cellules évoluent : une cellule blanche peut devenir noire et inversement, et ce, en fonction de son état et de l'état de ses deux cellules voisines, suivant une *règle* établie au départ. Un exemple de règle : une cellule se retrouve noire si, au pas de temps précédent, parmi elle et ses deux voisines il y avait un nombre impair de cellules noires (cette règle est appelée « rule 150 »). En fonction de la règle d'évolution, l'automate se comporte de manière drastiquement différente ; certaines règles donnent des résultats très répétitifs, d'autres complètement chaotiques.

Rule 110

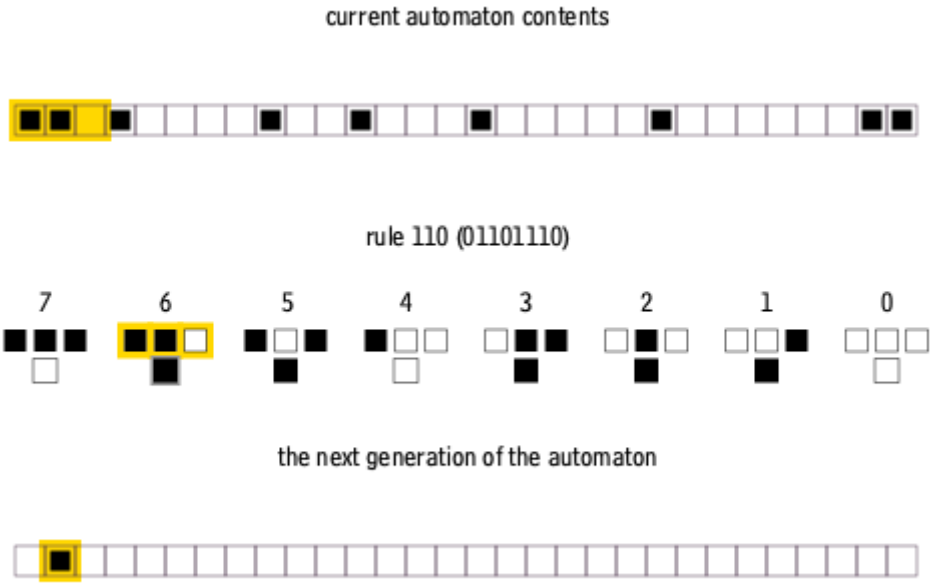

Illustration animée du fonctionnement de la règle 110. En haut, l'état du monde ; au milieu, la règle d'évolution ; en bas, la génération suivante. Source : Wikimedia Commons, CC-BY-SA

La règle 110 n'est pas très compliquée : une cellule ne peut se retrouver blanche que dans trois cas

- si elle et ses voisines étaient toutes trois blanches
- si elle et ses voisines étaient toutes trois noires
- si sa voisine de gauche était noire, mais elle-même et sa voisine de droite étaient blanches

Cette règle donne des résultats très intéressants, car à la fois réguliers et chaotiques : de fait, l'automate correspondant est capable de simuler l'exécution de n'importe quel programme (en encodant le programme comme un état initial du monde, et en lisant le résultat dans un état considéré comme final).

❦ Implémentation

Après ces éléments de contexte, passons à l'implémentation. On va écrire un script `rule-110.php`, destiné à être exécuté dans un terminal (ne cherchez pas à le mettre sur un serveur web, ce n'est pas le but ici).

Remarque : il est conseillé de mettre des types au paramètres et valeurs de retour des fonctions et méthodes, dans la mesure du possible. Il est également conseillé d'utiliser la vérification stricte des types. Voir cours.

Représentation du monde

On va créer une classe `WorldState`, qui représentera un état courant du monde. En interne, le monde sera représenté par un tableau de booléens, `true` représentant une cellule noire (vivante, présente) et `false` une cellule blanche (morte, absente). Depuis l'extérieur de la classe, on ne pourra pas accéder au tableau lui-même : on pourra seulement vérifier si une cellule à une position donnée est vivante ou non. En particulier, on ne pourra pas modifier l'état des cellules — les instances de `WorldState` sont dites *immutables*.

1. Créer une classe `WorldState` dans le script.
2. Lui déclarer une propriété `$cells`, de visibilité privée.
3. Le constructeur de la classe doit prendre un paramètre `$nbCells`, le nombre de cellules du monde (par défaut 100), et créer dans sa propriété `$cells` un tableau de booléens représentant les cellules, qui doivent toutes être mortes.
   Remarque : vous pouvez stocker `$nbCells` dans une propriété de la classe si vous voulez, mais ce n'est pas obligatoire, puisque l'on a accès à la longueur du tableau.
4. **Tester** : créer une *petite* instance de `WorldState` et l'afficher avec `var_export` ou `var_dump`. A-t-elle bien les propriétés attendues ?
5. Ajouter une méthode statique `buildFixedWorld($nbCells)`, qui va construire une instance de `WorldState`, rendre vivantes uniquement la deuxième et l'avant-dernière des cellules du tableau, et renvoyer cette instance. (Remarque : les méthodes de ce genre s'appellent des *factory methods*. On en ajoutera une autre plus tard.) Tester cette méthode de construction (de la même façon que précédemment).
6. Ajouter une méthode `isCellAliveAtPosition`. Elle doit attendre un entier `$position` en paramètre, et renvoyer `true` si et seulement si la cellule à la position indiquée est vivante (c'est-à-dire noire). NB: la position correspond à l'indice dans le tableau. Si la position demandée n'existe pas, lever une exception. Tester sur des cellules censées être mortes, sur des cellules censées être vivantes, et pour des positions trop grandes ou trop petites : assurez-vous dans tous ces cas que le comportement obtenu est le bon.
7. On veut pouvoir afficher le monde comme une chaîne de caractères, en utilisant des espaces pour les cellules blanches et un autre caractère pour les cellules noires (on pourra utiliser un #, ou le caractère unicode « FULL BLOCK, `&#9608;` »). Implémenter la méthode `__toString` afin de remplir cet objectif, et vérifier que l'affichage d'une instance construite par `buildFixedWorld` est bien cohérent.

Évolution

Pour implémenter l'évolution, on commence par une version simple mais peu flexible, qu'on rendra plus générale dans la suite.

1. Écrire une *fonction* `compute_next_state_rule110($leftAlive, $selfAlive, $rightAlive)`, qui renvoie l'état d'une cellule à la prochaine génération, en

fonction de son propre état (paramètre `$selfAlive`) et de celui de ses deux voisines (`$leftAlive` et `$rightAlive`).

2. Ajouter une méthode `computeNextGeneration()` à la classe `WorldState`, qui renvoie une nouvelle instance de `WorldState` représentant le prochain état du monde, après application de la règle d'évolution (grâce à `compute_next_state_rule110`). Pour simplifier, on ignorera les cellules aux extrémités du tableau (on considère que le monde est une ligne, et pas un cercle).
3. Écrire une classe `Simulator`, qui a comme propriété une instance de `WorldState` (passée à son constructeur), et comme méthode `displayEvolution($nbGenerations)`, qui affiche `$nbGenerations` successives du monde.

Pour tester le programme, construire un monde et un simulateur, et lui faire afficher 50 générations. Vérifiez que ça marche ! (La règle est-elle bien respectée à vue de nez ? Est-ce que ça ressemble aux dessins sur Wikipédia ?)

> Le TP étant très long, nous ne nous attendons pas à ce que cet exercice soit traité en intégralité par tou·tes les étudiant·es. Il est néanmoins vivement conseillé, dans la mesure du possible, de s'attaquer à la suite de l'exercice, qui aborde divers aspects de conception objet qui sont notamment pertinents dans le contexte du web (mais pas que). Si vous êtes assez à l'aise, vous devriez aller vite ; si vous n'êtes pas à l'aise, il est d'autant plus intéressant d'aller aussi loin que possible !

❦ Architecture pour les règles d'évolution

La façon dont la règle 110 a été implémentée dans le programme n'est pas très satisfaisante : il faut modifier le code de `WorldState` si on veut utiliser une autre règle. Pour éviter ça, une possibilité pourrait être de rendre `WorldState` abstraite et de lui ajouter une méthode abstraite `computeNextStateCell`, qui serait définie dans des sous-classes `WorldStateRule110`, `WorldStateRule150`, etc.

Cette solution est cependant un peu lourde, et limite la flexibilité du code. Une meilleure idée est d'utiliser la *composition* : `WorldState` va faire appel à un objet tiers pour le calcul de l'état suivant, et cet objet tiers pourra avoir diverses implémentations (un par règle). Ainsi la classe `WorldState` reste indépendante du changement de règle, et elle peut être elle-même modifiée sans risque d'impacter des sous-classes.

1. Créer une interface `EvolutionRule`, qui doit définir une seule méthode `computeNextStateCell($leftAlive, $selfAlive, $rightAlive)`.
2. Créer une classe `Rule110` qui implémente l'interface, et supprimer la fonction `compute_next_state_rule110`.
3. Modifier la méthode `computeNextGeneration` de `WorldState` : elle doit prendre en paramètre une instance de `EvolutionRule`, et utiliser sa méthode pour le

calcul de la génération suivante. NB: vous devez *forcer* le paramètre à avoir le type `EvolutionRule`.

4. L'instance de `EvolutionRule` va donc devoir être donnée par notre `Simulator`. A priori, c'est à la construction du `Simulator` que la règle va être choisie. Par conséquent, ajouter une propriété `$evolutionRule` à la classe `Simulator`, propriété qui doit être initialisée via un paramètre du constructeur. Modifier ensuite l'appel à `computeNextState`.
5. Tester, en donnant au simulateur une instance de `Rule110` : le programme doit fonctionner comme avant.

À présent, pour utiliser une autre règle, il suffit de créer une implémentation différente de `EvolutionRule`.

1. **Optionnel :** Implémenter une autre règle, par exemple la règle 184, et tester. Voir ici des dessins d'un bon nombre d'autres règles.

❦ Architecture pour l'affichage

L'utilisation d'une méthode `__toString` pour la partie « affichage » du programme est pratique, mais pas très robuste : on est dépendant de l'implémentation de `__toString` (on ne peut pas décider de changer les caractères, ou la taille des cellules, depuis l'extérieur de la classe), et réciproquement, on ne peut pas faire évoluer `__toString` n'importe comment. Cette partie vise à vous montrer ça.

Modification du `__toString`, et impact sur le programme

1. On va ajouter une propriété `age` à `WorldState`, qui sera initialisée à 0 dans le constructeur, et incrémentée dans `computeNextGeneration` comme il se doit. Elle aura bien sûr la visibilité privée, et on lui ajoutera un accesseur (mais pas de mutateur — on veut toujours que `WorldState` soit immutable).
2. Modifier le `__toString` pour que l'âge du monde apparaisse entre parenthèses au début de la chaîne. Relancer le programme pour voir le résultat.

Notre nouvelle méthode `__toString` est pratique pour faire des tests sur le programme, mais elle impacte l'affichage des simulations : ce n'est pas (forcément) ce qu'on veut.

De manière générale il est une bonne idée de séparer le *modèle*, ou *logique métier*, du programme, et l'affichage, qui peut avoir besoin de varier en fonction des contextes. Nous allons nous y employer.

Séparation modèle et vue

1. Créer une interface `Displayer` avec une seule méthode, `displayWorld`, qui doit

prendre en paramètre une instance de `WorldState`.

2. Modifier `Simulator` pour qu'il utilise une instance de `Displayer` pour afficher le monde, plutôt que de faire un `echo`.
3. Créer `TerminalDisplayer`, l'implémentation de `Displayer` qui correspond à l'affichage qu'on avait auparavant (sans l'âge). NB: on peut passer en paramètre du constructeur de `TerminalDisplayer` les caractères à utiliser pour les cellules blanches et noires ! Ce n'était pas vraiment possible de le faire quand on utilisait le `__toString` de `WorldState` sans alourdir futilement l'API de la classe.
4. Passer une instance de `TerminalDisplayer` au simulateur, et tester que le programme fonctionne bien comme avant.

Contrôle plus fin de la vitesse des itérations

Notre programme affiche l'évolution du monde, mais si on veut afficher un grand nombre d'itérations, le défilement peut être très rapide. On voudrait pouvoir le gérer plus finement : soit en faisant une petite pause après chaque affichage, soit en attendant que l'utilisateurtrice appuie sur Entrée avant de passer à la génération suivante.

1. Ajouter une méthode *abstraite* `iterationControl()` à `TerminalDisplayer` (qui va donc devoir être une classe abstraite). Cette méthode doit être appelée à la fin de `displayWorld`.
2. Créer une sous-classe `InteractiveTerminalDisplayer` qui implémente `iterationControl` en attendant l'appui sur Entrée (rappelez-vous de la fonction `readline` vue au premier TP), et tester le résultat.
3. Créer une autre sous-classe, `PausingTerminalDisplayer`, qui implémente `iterationControl` en attendant un certain nombre de microsecondes (éventuellement paramétrable) (voir la fonction usleep — qui ne marche pas forcément sous windows). Tester le résultat.
4. **Optionnel** : créer une classe `AnimatedTerminalDisplayer` qui **étend `PausingTerminalDisplayer`**. Elle doit appeler le `iterationControl` de sa parente, puis afficher la chaîne suivante : `"\e[2J\e[1;1H"`. Il s'agit de séquences d'échappement qui effacent le contenu du terminal (pour les terminaux compatibles — ça devrait toujours marcher sous Linux, sans doute aussi sous OSX et BSD, et pour Windows ça dépend des versions). Ce *displayer* permet donc de voir l'évolution du monde de façon animée (ça marche mieux si les itérations ne sont pas trop rapides, typiquement 3-5 FPS). Tester le résultat.

Un affichage différent

1. Créer une classe `StatsDisplayer`, qui implémente `Displayer` en montrant non pas l'état du monde, mais des statistiques dessus : âge et nombre de cellules noires/blanches. Tester.

2. Comment faire pour avoir l'affichage des statistiques avec un défilement lent / interactif, comme on l'a fait dans la section précédente ? Les choix d'architecture pour le contrôle de la vitesse des itérations étaient-ils bons ? Comment aurait-on pu faire ?