# R3.01 - Développement Web - PHP

Comme de nombreux autres langages,
PHP s'est fortement inspiré du langage C.
Sa syntaxe en est donc proche sur de nombreux aspects :

- Langage procédural et Objet
- Instructions séparées ou terminées par `;`
- Blocs de code encadrés par des `{}`
- Cellules des tableaux accédées par des `[]`
- Mots clés du langage : `if`, `for`, `while`, `switch`, `print`, `function`, `return`, `include`, ~~`goto`~~ (oups pardon)

Nous ne nous arrêterons que
sur ce qui diverge vraiment dans PHP
et ce qui en fait sa spécificité.

# Qu’est-ce que PHP ?

- Langage de scripting (PHP, Python, Ruby, Perl, Bash, ...)
- Créé par Rasmus Lerdorf en 1995 (Version 1)
- Intimement lié au monde du développement Web
- Exclusivement côté serveur
- PHP = Personal Home Page puis PHP Hypertext Preprocessor
- Version 8 depuis 2020
- Version < 8 = obsolètes. Uniquement des màj de sécurité.
- Deux contextes : page Web ou script autonome (cf R1.04)

# Principaux concurrents

- Javascript
- Java
- Ruby
- Python
- Rust
- Go
- C#

# Code PHP

- Page Web PHP = mix HTML et code PHP
- Encapsulation du PHP : `<?php ... ?>`
- Hors tags = texte (HTML ou autre). Traité comme des instructions `echo`
- Possibilité de zones de code PHP multiples dans une même page.
- Extension (page Web) : `.php`
- Interpréteur : module serveur (Apache, nginx) pour pages Web ou binaire autonome pour scripts (vu en R1.04)

# Exemple de page Web avec du PHP

```
<html>
   <head>
      <title>Hello World</title>
   </head>
   <body>
<?php
   echo "Hello World!"
?>
La somme des 100 premiers nombres = <?php echo (100*101)/2 ?>
   </body>
</html>
```

Envoyé en l'état vers navigateur sans aucune interprétation par PHP

**En rouge** : partie interprétée.
Résultat envoyé vers navigateur

# Exemple de MAUVAIS code PHP

```
<?php
echo "<html>\n<head>\n<title>Hello
World</title>\n</head>\n<body>\nHello World!\n
La somme des 100 premiers nombres = " .
(100*101/2) . "\n</body>\n</html>"
?>
```

# Conseil

Du code HTML en dehors des tags `<?php ... ?>` permet :

- Coloration syntaxique dans éditeur (VSC, etc.)
- Évite l'interprétation sans intérêt des zones de texte (HTML)

Perler le code HTML de portions de PHP :

- Pas d'impact notable sur les performances
- PHP est conçu pour fonctionner ainsi
- Plus lisible qu'un gros bloc de `echo`

# Avantages

- Base de développeurs très solide
- Librairies (extensions) très nombreuses et variées
- Nombreux frameworks
- Simplicité et syntaxe

# Inconvénients

- Très permissif
- Seulement côté serveur

# Types (principaux)

## Scalaires :

- Entier
- Réel
- Booléen
- Chaîne

## Autres :

- Tableau
- Objet
- Ressource (fichiers)
- null
- true / false

# Variables

- Ne sont pas définies avant leur affectation
- N'ont pas de type défini à l'avance
- Ont un type implicite déterminé par l'affectation (voir opérateur d'affectation plus loin)
- Ont un type qui peut évoluer si elles sont réaffectées
- Ont un nom sensible à la casse (min/MAJ) et préfixé d'un $
  Par convention, éviter les noms en MAJ (réservé aux constantes)
- Peuvent être supprimées (espace mémoire libéré)

# Syntaxe des variables

## Affectation : $<nom> = <val>;

- Entier : $age = 12;
- Réel : $prix = 5.75;
- Booléen : $present = false;
- Chaîne : $nom = "Toto";

## Usage : $<nom>

- echo $age;
- $ttc = $prix * $taux_tva;

# Portée des variables

La portée = la visibilité et l'accessibilité.
Par défaut, les variables :

- Définies dans une fonction, lui sont locales et ne sont pas accessibles non plus dans les sous-fonctions s'il y en a.
- Définies globalement (dans le script, en dehors d'une fonction), ne sont pas visibles non plus dans les fonctions !
- **ATTENTION** : comme PHP accepte d'affecter une variable sans la définir avant, il est facile de penser modifier une variable globale alors qu'on en crée une locale du même nom !

# Portée globale des variables

Solution : le mot-clé `global`

Permet :

- De définir, dans une fonction, qu'on veut accéder (lecture ou écriture) à une variable définie en dehors de la fonction.
- De définir localement une nouvelle variable globale, qu'on souhaite rendre visible (et donc statique) à l'extérieur de la fonction. Les autres fonctions qui veulent la voir devront elles-mêmes signifier cela avec un mot-clé `global` aussi.

# Portée globale des variables

## Exemple :

```
$ville = "Lannion";
function maVille() {
    global $ville;
    global $compteur;
    echo "J'habite $ville\n";
    $compteur++;
}
maVille();
maVille();
echo "On a appelé $compteur fois la fonction maVille\n";
```

← On peut maintenant accéder à la **globale `$ville`**

← Cette variable sera maintenant une nouvelle variable **globale**, donc visible en dehors de la fonction. Valeur par défaut à **0**.

# Constantes

- Ont les mêmes caractéristiques que les variables mais…
- Sont affectées une seule fois et ont donc un type inaltérable
- Sont définies de 2 façons possibles :

```
define("PI", 3.14);
const PI = 3.14;
```

- Par convention, les noms sont en MAJ
- Ont une portée globale (pas comme les variables)
- **ATTENTION** : on peut modifier les valeurs dans un tableau “constant”. C’est sa nature du “tableau” qui est constant.

# Chaînes de caractères

Les chaînes peuvent être définies de plusieurs façons.

Deux principes :

- Chaînes non interprétées : entre `'...'`
  Le contenu est conservé en l'état
- Chaînes interprétées : entre `"..."`
  PHP va identifier et convertir certains contenus comme les variables (`$nomvar` ou `${nomvar}`), les caractères de contrôles (`\n`, `\t`, etc.)

← A privilégier pour du texte brut, pas d'interprétation, c'est plus rapide et sans effet indésirable ou non maîtrisé.

# Chaînes de caractères

## Exemple :

```
$nom = "PEUPLU";
$prenom = "JEAN";
echo "Bonjour $nom $prenom\n";
echo 'Bonjour $nom $prenom\n';
```

← Affiche `Bonjour JEAN PEUPLU`

← Affiche `Bonjour $nom $prenom\n`

# Debogage des variables

- `print_r($var)` Affiche (sur STDOUT) le contenu d'une variable. Capacité d'affichage récursif dans le cas de tableaux.
- `var_export($var)` Similaire à `print_r()`
- `var_export($var, true)` Renvoie une chaîne au lieu d'afficher (pratique pour logguer dans un fichier)
- `var_dump($var)` Similaire à `print_r()` mais avec des informations sur le type de la variable.

**Astuce** : dans une page Web placer l'appel entre des tags HTML `<pre>...</pre>` pour rendre plus lisible.

# Tableaux

Les tableaux sont très utilisés en PHP.

Il existe de nombreuses fonctions de manipulation des tableaux.

En R1.04 vous avez eu l'occasion d'en expérimenter quelques unes, comme `file_get_contents()` qui permet la lecture d'un fichier et son stockage ligne par ligne dans les cellules d'un tableau.

# Tableaux indicés

Les tableaux indicés sont similaires à ceux qui existent en C :

- Indices de `0` à `N-1` où `N` est la taille du tableau

Mais en PHP, les tableaux :

- Sont dynamiques : grandissent au fur et à mesure des ajouts
- Peuvent stocker tout type de donnée
- Peuvent stocker des types différents dans un même tableau
- Peuvent stocker d'autres tableaux : c'est ainsi qu'on peut définir des tableaux multi-dimension

# Tableaux indicés

Définition et initialisation :

```
$tablo = [
    "Lannion",
    "St Brieuc",
    "Guingamp"
];

print_r($tablo);
```

→ Affichage du `print_r($tablo)`

```
Array (
    [0] => Lannion
    [1] => St Brieuc
    [2] => Guingamp
)
```

On a bien un tableau indicé `0..2`

Mais on y revient un peu plus loin...

# Tableaux associatifs

Les tableaux associatifs permettent d'associer chaque cellule du tableau à une clé unique d'un type scalaire quelconque (entier, réel, chaîne).

PHP propose aussi ce type de tableau.

PHP ne propose d'ailleurs QUE ce type de tableau !!!

Les tableaux indicés ne sont qu'un maquillage de tableau associatif où les indices (clés) sont simplement des entiers de `0` à `N-1`...

# Tableaux associatifs

Définition et initialisation :

```
$tablo = [
  22300 => "Lannion",
  22000 => "St Brieuc",
  22200 => "Guingamp"
];

print_r($tablo);
```

→ Affichage du `print_r($tablo)`

```
Array (
    [22300] => Lannion
    [22000] => St Brieuc
    [22200] => Guingamp
)
```

# Manipulation de tableaux

- Ajout de cellules :

```
$tablo[] = …;
$table[<clé>] = …;
array_push($tablo, <val>, …);
```

- Suppression de cellules :

```
unset($tablo[<clé>]);
```

- Voir les fonctions `array_xxx()` sur le site php.net
- Parcours des cellules : voir `foreach()` plus loin (boucles)

← **clé** calculée comme `MAX(clés) + 1`

← Si la **clé** existe, remplace la **valeur**, sinon ajoute le nouveau couple **clé-valeur**
← Ajout en fin de tableau sur le même principe de calcul du `MAX(clés) + 1`

← `unset()` fonctionne aussi sur les autres types de variable. Provoque des **trous** dans les “indices”, preuve que ce ne sont pas des indices mais des clés !

# Opérateurs

- Affectation
- Arithmétiques
- Incrément / Décrément
- Logiques
- Comparaison
- Chaînes
- Bits

# Opérateurs

PHP est très souple et permissif (attention).

En fonction du contexte (opérateur utilisé), PHP convertit les types (variables, expressions) pour satisfaire au résultat attendu :

- "12.5" * 2 = 25 (réel) → "12.5" est converti en réel pour faire l'opération
- "_" . 12.5 . "_" = "_12.5_" (chaîne) → 12.5 est converti en chaîne pour concaténer
- "1e3" / 2 = 500 → "1e3" est converti en 1000 (car PHP l'identifie comme une écriture scientifique de 1000)

# Opérateurs d'affectation

L'affectation :

- Deux opérandes
- Opérande de gauche est une variable
- Opérateur d'affectation "=”
- Opérande de droite est une expression. Il détermine aussi le type de la variable qui va être affectée

Voir les opérateurs d'incrémentation et décrémentation plus loin.

# Opérateurs d'affectation

Exemples :

- `$codep = 22300;` → Type entier
- `$lannion = $codep;` → Type entier
- `$pi = 22/7;` → Type réel
- `$nom = 'Toto';` → Type chaîne
- `$arr = [];` → Type tableau (vide)
- `$lieu = "$codep Lannion";` → Type chaîne

# Autres opérateurs d'affectation

- `$var+=<val>` Ajoute `<val>` à la valeur de `$var` et réaffectation du résultat à `$var`
- `$var-=<val>` Idem avec une soustraction
- `$var*=<val>` Idem avec une multiplication
- `$var/=<val>` Idem avec une division
- `$var%=<val>` Idem avec un modulo (reste de division)

# Opérateurs arithmétiques

## Entiers et réels :

- Addition : ... + ...
- Soustraction : ... - ...
- Multiplication : ... * ...
- Division : ... / ...
- Exponentiation (puissance) : ... ** ...

## Entiers :

- Reste de la division euclidienne (modulo) : ... % ...

# Opérateurs arithmétiques

Type du résultat (entier ou réel) est fonction des opérandes et/ou de la nécessité de ne pas perdre en précision. Exemples :

- `4/2` → Résultat entier (2 opérandes entiers)
- `5/2` → Résultat réel (sinon perte de précision)
- `4.0/2` → Résultat réel (conservation du type le plus “précis”)
- `5%2` → Résultat entier (domaine de validité = entiers seulement)
- `5.0%2` → Résultat entier (domaine de validité = entiers seulement)
- `5.5%2` → Résultat entier (domaine de validité = entiers seulement)

# Opérateurs arithmétiques

Entiers et réels :

- Identité : `+` ...
- Négation : `-` ...

Entiers :

- Division entière : `intdiv(... , ...)`

Astuce : l'identité devant une expression → force la conversion.

Exemple : `"123"` → Chaîne, mais `+"123"` → Entier, et `+"123.0"` → Réel

# Opérateurs d'incrément / décrémentation

Ce sont aussi des opérateurs d'affectation.

- `$var++` Post-incrémentation de 1 (incrémentée après son évaluation dans l'expression)
- `++$var` Pré-incrémentation de 1 (incrémentée avant son évaluation dans l'expression)
- `$var--` et `--$var` : idem en décrémentation de 1

# Opérateurs logiques

## Opérations booléennes :

- ET : … `and` …
- ET : … `&&` …
- OU : … `or` …
- OU : … `||` …
- OU exclusif : … `xor` …
- NON : `!`…

Ne pas hésiter à mettre des `()` pour **expliciter sans ambiguïté** ce qu'on veut faire !

```
$a || $b && $c || $d && $e
```

se lit nettement mieux ainsi :

```
$a || ($b && $c) || ($d && $e)
```

même si le résultat est le même !

**Note : ne pas confondre `||` et `&&` avec les opérateurs bits `&` et `|`**

# Opérateurs de comparaison

Attention, il y a des subtilités, surtout du fait de la manière dont PHP gère les types et de sa permissivité en la matière.

- `==` Égalité au sens large (au sens PHP).
  Ainsi : `1 == 1.0` ou encore `1 == "1.0"` évaluent à vrai
- `===` Identité : égalité stricte (valeur et type)
  Ainsi : `1 === "1"` évalue à faux
- `!=` ou `<>` Différence au sens large
- `!==` Différence stricte (valeur ou type)

# Opérateurs de comparaison

- **<**, **<=**, **>**, **>=**, les comparaisons classiques.
  Fonctionnent sur les valeurs numériques
  Fonctionnent sur les chaînes.
  Attention : avec chaînes → comparaison numérique si possible.
  Ainsi :
  - `"ABC" < "ABD"` est vrai
  - `"abc" < "ABD"` est faux (min après MAJ dans table ASCII)
  - `"12" < "110"` est vrai (serait faux avec `strcmp()` en C)

# Opérateurs de comparaison

- <=> comparaison combinée

  Fonctionne sur les valeurs numériques

  Fonctionne sur les chaînes.

  Comparaison au sens large

  Renvoie :

  - -1 si c'est <
  - 0 si c'est ==
  - 1 si c'est >

# Opérateurs de chaînes

- `.` (point) concaténation.

```
$login = $nom . "_" . $groupe_tp;
```

- `.=` concaténation + affectation (ajout à la fin)

```
$login .= "_1A1";
```

- Variables dans les chaînes entre `"..."` :

```
$login = "${nom}_${groupe_tp}";
```

Les `{}` sont nécessaires si risque de mauvaise interprétation :

`"$nom 1A1"` → OK.

`"$nom_1A1"` = KO (→ `var nom_1A1`)

# Opérateurs bits

- `|` ou binaire (ne pas confondre avec le `||`, ou logique)

  `4 | 5` → `100` (4 en binaire) `|` `101` (5 en binaire) `= 5` (bit à bit)

  `4 || 7` → `TRUE` (4 équivaut à `TRUE` en bool) `||` `TRUE` (7 équivaut à `TRUE` en bool) `= TRUE`
- `&` et binaire (ne pas confondre avec le `&&`, et logique)
- `^` ou exclusif

  `4 ^ 7` → `100 ^ 111 = 3` (bit à bit)
- `~` négation binaire (ne pas confondre avec le `!`, non logique)

  `~4` → `3` (`NON 100` → `011`, bit à bit)

# Structure de contrôle - Test

- Test simple

```
if <expression_logique> {
…
}
```

- Test avec une alternative

```
if <expression_logique> {
…
} else {
…
}
```

Les **{ }** sont optionnelles si une seule action à exécuter.

Mais il est conseillé de toujours les utiliser.

# Structure de contrôle - Test

- Test avec plusieurs alternatives

```
if <expression_logique> {
…
} elseif <expression_logique> {
…
} else {
…
}
```

← Les **elseif** peuvent être multiples. On peut aussi écrire **else if**

← Le **else** final est optionnel.

# Structure de contrôle - Test

## Autre syntaxe sans { } :

```
if <expression_logique>:
...
elseif <expression_logique>:
...
else:
...
endif;
```

← On peut avoir plusieurs instructions dans chaque bloc.

# Structure de contrôle - Test

## Autre syntaxe avec `switch` :

```
switch <expression> {
    case VAL1:
        ...
        break;
    case VAL2:
        ...
        break;
    default:
        ...
        break;
}
```

← On teste sur des expressions constantes. On évitera d'utiliser des variables ici.

← Le **`break`** est optionnel. Si absent, le code du **`case`** suivant est aussi exécuté (même si la condition de cet autre **`case`** n'est pas remplie), jusqu'à rencontrer une instruction **`break`** ou **`continue`**.

← Le **`default`** est exécuté si aucun **`case`** n'a été trouvé. Il est optionnel. Généralement on omet son **`break`** si on place **`default`** après tous les **`case`**.

# Structure de contrôle - Opérateur ternaire

Il s'agit autant d'un opérateur que d'une structure de contrôle.

C'est un `if else` raccourci.

Syntaxe :

```
$statut = ($age < 18) ? "Mineur" : "Majeur";
```

# Structure de contrôle - Boucle “for”

## Syntaxe :

```
for (<expression_initialisation>;
     <condition_bouclage>;
     <instruction_avant_bouclage>) {
...
}
```

Les **{ }** sont optionnelles si une seule action à exécuter.

Mais il est conseillé de toujours les utiliser.

# Structure de contrôle - Boucle “for”

## Exemples :

```
for ($loop = 1, $var = ""; $loop < 10;
     $loop += 2) {
...
}
```

← Chacune des 3 parties du **`for()`** peut contenir plusieurs instructions, séparées par des virgules (car le `;` sépare déjà les 3 parties)
Note : en langage C aussi…

```
for ( ; $nom !== "" ; ) {
...
}
```

← Les 3 parties de l’instruction **`for()`** sont toutes optionnelles.

Ainsi **`for(;;)`** est une boucle sans fin.

# Structure de contrôle - Boucle “while”

## Syntaxe pré-condition :

```
while (<condition_bouclage>) {
...
}
```

Les { } sont optionnelles si une seule action à exécuter.

Mais il est conseillé de toujours les utiliser.

## Syntaxe post-condition :

```
do {
...
} while (<condition_bouclage>);
```

← On passe obligatoirement au moins une fois dans la boucle.

# Structure de contrôle - Boucle “foreach”

Utilisé avec les types “itérables”, comme les tableaux ou les classes implémentant l’interface Traversable (ne sera pas étudié, consulter la doc sur php.net si besoin).

Syntaxe pour parcourir les **valeurs** d’un tableau :

```
foreach ($tablo as $val) {
...
}
```

← A chaque passage, **`$val`** contient la valeur d’une cellule.

# Structure de contrôle - Boucle “foreach”

## Syntaxe pour parcourir les valeurs et les clés d’un tableau :

```
foreach ($tablo as $cle => $val) {
...
}
```

← A chaque passage, **`$val`** contient la valeur d’une cellule et **`$cle`** la clé qui lui est associée.

Cas d’un tableau indexé : clés vont de **0** à **`N-1`**

# Structure de contrôle - Boucle “foreach”

## Si besoin de la modifier la valeur, utiliser la syntaxe :

```
foreach ($tablo as $cle => &$val) {
    $val = ...;
}
```

← Le **&** signifie “**référence**”, une sorte de pointeur du langage C, mais avec une syntaxe plus simple.

La clé est **inaltérable**, on ne peut donc pas la préfixer d’un **&** pour espérer la modifier.

# Inclusion de code

Placé généralement en tête des scripts.

Permet d'insérer dans son code, le code contenu dans un autre fichier. Similaire au `#include` du langage C.

- **`require_once`** : code inséré une seule fois, même si le même `require_once` apparaît à plusieurs endroits du code.
  Erreur et fin d'exécution si pb rencontré pour insérer le code.
- **`require`** : code inséré à chaque endroit où `require` apparaît.
  Erreur et fin d'exécution si pb rencontré pour insérer le code.

# Inclusion de code

- **`include_once`** : code inséré une seule fois, même si le même `include_once` apparaît à plusieurs endroits du code.
  Simple avertissement si pb rencontré pour insérer le code.
- **`include`** : code inséré à chaque endroit où `include` apparaît.
  Simple avertissement si pb rencontré pour insérer le code.

Syntaxe : `require_once "../lib/toolz.php";`

# Fonctions utilisateur

- Peuvent être définies à peu près partout dans un script, y compris dans une autre fonction !
- Les fonctions définies dans d'autres fonctions doivent être définies avant leur appel (i.e. la fonction parente doit avoir été appelée)
- Ne peuvent pas être redéfinies
- Ne peuvent pas être surchargées (plusieurs signatures pour un même nom)
- Ne peuvent pas être supprimées une fois définies

# Définition

```
function bonjour($nom, $prenom, $age) {
    echo "Bonjour $nom $prenom\n";
    if (!$age) {
        echo "Je ne saurais vous donner un âge…\n"
    }
}

bonjour("Mensoif", "Gérard");
echo bonjour("Padmandé", "José", 37);
```

PHP est très souple (trop ?) :

← **`$age`** vaut **`null`**, car pas spécifié à l'appel.
← N'affiche rien car **`bonjour()`** ne retourne rien (en fait elle retourne **`null`** si pas de mot-clé **`return`**)

# Fonctions imbriquées

```
function outer() {
    function inner() {
        echo "Bonjour je suis inner\n";
    }
    echo "Bonjour je suis outer\n";
}

// Impossible d'appeler inner()
outer();
inner();
```

PHP est très souple (trop ?) :

- **`inner()`** est encore indéfinie
- L'appel de **`outer()`** engendre la définition de **`inner()`**
- L'appel de **`inner()`** est maintenant possible

# Retour

## Une fonction peut renvoyer plusieurs types. C'est assez commun :

```
function racine($val) {
    if ($val < 0) {
        return false;
    } else {
        return sqrt($val);
    }
}
```

← Booléen **`false`** pour signifier une erreur.

← Le résultat (ici un réel) si OK.

**IMPORTANT** il faudra tester le résultat avec un **`=== false`** pour détecter une erreur car **0** et **`false`** sont équivalents avec **==**, or **0** est une valeur correcte ($\sqrt{0} \rightarrow 0$)

# Retour multiple

## Une fonction peut renvoyer plusieurs valeurs. C'est puissant !

```php
function coord() {
    return [48.758334,-3.451388];
}

list($lat, $lng) = coord();
```

← Renvoie un tableau de valeurs

← **`list()`** permet de déstructurer le tableau renvoyé en alimentant des variables. Si pas assez de valeurs, on obtient des **`null`** à la place. C'est un mot-clé du langage.

# Arguments par référence

Par défaut, les arguments passés à une fonction le sont par valeur. Changer la valeur d'un argument dans la fonction n'impacte pas la variable originelle.

Passage par référence : permet le changement de la source :

```
function vider(&$nom) {
    $nom = "";
}
vider($toto);
```

← C'est à la définition de la fonction qu'on décide du mode de passage (**par valeur** ou, comme ici, **par référence** avec un **&**)

← Aucun changement pour l'appel. Mais **par référence** nécessite obligatoirement une variable.

# Les extensions / librairies

PHP dispose d'une panoplie exceptionnelle de fonctions en tout genre. Ce sont des extensions qu'il est possible d'activer/désactiver en fonction des besoins.

Il serait fastidieux et inutile de les lister toutes ici.

Source incontournable : php.net

**Astuce** : aller faire un tour de temps en temps sur ce site pour explorer les familles de fonction. Ca permet de garder en mémoire qu'il existe des choses pour faire tel ou tel traitement.

# Le modèle objet

PHP n’est pas né Langage Objet

L’implémentation du modèle objet s’est faite à la sauce PHP, avec des spécificités un peu atypiques par rapport aux autres langages Objet.

Sera vu en TD/TP

# Cookies et Sessions

Ne fonctionnent que dans un contexte de page Web.

Les informations associées à une session restent stockées sur le serveur Web.

Seul un cookie de session transite sur le réseau.

# Sessions

Démarrage d'une session ou reprise d'une session existante :

- Nouvelle session ou reprise ? Détection automatique par PHP
- `session_start()`
- Écrit des informations dans l'entête HTTP (rien à voir avec le `<head>` de la page HTML !), on est au niveau protocol HTTP (on en reparlera en R3.06)

  ➜ Doit être placé le plus tôt possible dans le script, avant tout autre affichage. C'est impératif.

# Sessions

Stockage et accès aux variables de session :

- Nécessite au préalable un appel réussi à `session_start()`
- Accès aux variables de session par le tableau (superglobale) `$_SESSION[]` qui se manipule comme tout tableau PHP.
- Stockage dans ce tableau sous forme clé-valeur : `$_SESSION[<clé>] = …;`
- Suppression d'une variable : `unset($_SESSION[<clé>])`

# Sessions

Destruction d'une session (à faire dans ce sens) :

- Appel à `session_unset()` Suppression du tableau `$_SESSION`
- Appel à `session_destroy()` Destruction de la session
- Le cookie est préservé, même si la session n'existe plus.
- Le cookie peut servir à recréer une autre session

# Les superglobales

Tableaux globaux entretenus en interne par PHP :

- `$GLOBALS` référence toutes les autres variables, y compris elle-même ! Tester avec un `print_r($GLOBALS)`
- `$_SESSION` variables de la session (vu précédemment)
- `$_GET` paramètres de l'URL ou d'un `<form>` (`method GET`)
- `$_POST` paramètres provenant d'un `<form>` (`method POST`)
- `$_FILES` infos si envoi de fichier dans un `<form>`
- `$_SERVER` infos utiles sur le script et l'environnement serveur
- `$_ENV` variables d'environnement (plutôt mode script autonome)