

はじめに

これから学ぶこと

本課題は、プログラム言語の1つである「PHP」を学びます。

基本的には、[PHPの公式言語リファレンス](#) の要約になりますので、何かあればそちらを参照してください。

これから学習する内容についてですが、これらは**全てではなく基礎的なものだけ**になっています。

もし、あなたが大きなサービスやシステムを構築する場合、より多くの知識が必要になるでしょう。

本課題が、それらを学ぶ足がけになるようであれば幸いです。

さらに詳しく知りたい場合や、よく分からない場合には是非、検索をしてみたり、本を読んでみたり、有識者に話を聞くなどしてみましょう。

前提条件

まず、知識としては「いきなり はじめる PHP」を一通り完了していることとします。

使用言語は「**PHP**」、バージョンの指定は行いません。

実行環境については、PHPが動き、結果が確認できる環境であれば、問題ありません。

PHPについて

本課題では、**PHP**のコーディングの基本を学んでいきます。

PHPは**サーバサイド**の基本的な処理に使用される言語で、その学習のしやすさから多くの人に使われています。

また、PHPは、以下のような特徴を持っています。

- HTMLに埋め込みが可能である。
- 型（中身の種類を決めるもの）の判定を自動でやってくれる。

これらの特徴から初心者に向いていると言われています。

またPHPは、**WEBアプリケーション、サービスに作るのに適している言語**です。

PHPでプログラミングしていく場合には、`<?php ?>`タグの中に処理を記述していきます。

```
<?php

// 変数の宣言
$basic_php = 'PHPの基礎だよ!';

// var_dump(確認したい変数)
var_dump($basic_php) // PHPの基礎だよ!

?>
```

また、ソース上に **コメント** を残したい場合、

// を文頭につけることで**実行されない文字列を残す**ことができます。

なので、上記のサンプルコード上の

```
// 変数の宣言
```

は実行されません。

1. 型

変数宣言時、宣言方法によって値の種類を見分ける**型**が決まります。

型によってできることが違い、それぞれの役割があります。

＊「型」 ([PHP公式言語リファレンス](#)) [🔗](#)

論理値 (Boolean)

論理値型(Boolean) は、最も簡単な型で**真偽の値**を表します。

■ 論理値型の宣言

値は、**TRUE** または **FALSE** のどちらかになります。

```
$true = True;  
$false = False;  
  
var_dump($true); // bool(true)  
var_dump($false); // bool(false)
```


整数 (Integer)

整数型 (Integer) は数字を扱う型です。

■ 整数型の宣言

変数宣言時には整数を代入します。

なお、**整数型では少数点が扱えません**。

少数点を扱いたい場合には、[浮動小数点数型](#)  を参照してください。

```
$number = 6;  
var_dump($number); // int(6)
```

■ 演算子を使用した計算

数値型では演算子を使用した計算が可能です。

```
// 整数の宣言  
$a = 9;  
$b = 3;  
  
// 加算 $a および $b の合計  
$c = $a + $b;  
var_dump($c); // int(12)  
  
// 減算 $a と $b の差  
$c = $a - $b;  
var_dump($c); // int(6)  
  
// 乗算 $a および $b の積  
$c = $a * $b;  
var_dump($c); // int(27)  
  
// 除算 $a および $b の商  
$c = $a / $b;  
var_dump($c); // int(3)  
  
// 剰余 $a を $b で割った余り  
$c = $a % $b;  
var_dump($c); // int(0)  
  
// 累乗 $a の $b 乗。 PHP 5.6から。  
$c = $a ** $b;  
var_dump($c); // int(729)
```

文字列 (String)

文字列 (String) はその名の通り文字列を扱う型です。

文字列型では文字列間の結合や、文字の置換などが行えます。

画面への表示も文字列での実行になるので扱う機会は多くあるでしょう。

■ 文字列型の宣言

文字列を**クォーテーション**で囲って変数に代入します。

また、ダブルクォーテーションでも構いません。

```
$basic_php = 'PHPの基礎だよ!';  
  
var_dump($basic_php); // string(PHPの基礎だよ!)
```

■ 文字列の結合

変数を**ドット**で繋ぐことで文字列の結合が可能です。

```
// 文字列型の変数宣言  
$word = "文字列";  
$join = "結合";  
  
// ドット(.)で繋ぐ  
$word_join = $word.$join;  
  
var_dump($word_join); // string(文字列結合)
```

また、文字列と変数でも可能です。

```
$word = "文字列";  
$joined_word = '結合した'.$word;  
  
var_dump($joined_word); // string(結合した文字列)
```


配列 (array)

さて、今までは**1つの変数**に対して**1つの値**を格納していました。
次に**1つの変数に複数の値**をもつ型を学んでいきます。それが**配列型**です。

■ 配列の宣言

配列の宣言には、**array()** を使用します。

```
// array()を使用した宣言
$array = array( 'data1', 'data2', 'data3' );

var_dump($array); // ['data1','data2','data3']
```

■ 配列内の値の取得

配列の宣言時、値それぞれに**0から順に数字（キー）**が振られます。
特定の値を取得したい場合はこの**キーを指定**し、値を**取得**します。

```
$array = array(
    'data1',    // 0 ← キー
    'data2',    // 1
    'data3'     // 2
);

//キー指定で値の取得を行う
var_dump($array[0]); // 'data1'
```

■ 配列内の値の追加

配列型は **\$変数[]** に値を代入することで配列の追加が可能です。

```
$array = array(
    'data1',    // 0 ← キー
    'data2',    // 1
    'data3'     // 2
);

// 配列の追加
$array[] = 'data4';

//追加されている
var_dump($array); // ['data1', 'data2', 'data3', 'data4', ]

//キー指定で値の取得を行う
var_dump($array[3]); // 'data4'
```

連想配列

また配列の値を取得するためのキーは、**自分で決めた文字列を設定することも可能**です。

これを**連想配列**と呼びます。

配列と同じようにarray()で宣言しますが、その際に**キーを指定**します。

■ 連想配列の宣言

```
// 'キー' => データ で宣言
$array = array(
    'one' => 'data1',      // 'one' ← キー
    'two' => 'data2',      // 'two'
    'three' => 'data3'     // 'three'
);
```

■ 連想配列の値の取得

値の取得をしたい場合には、配列と同じようにキー指定を行います。

```
// キー指定で値の取得を行う
var_dump($array['two']); // 'data2'
```

■ 連想配列の値の追加

値を追加したい時には、**キーを指定し**、値を代入します。

また、**すでに存在しているキーを指定した場合には値が上書きされるので**注意しましょう。

```
$array = array(
    'one' => 'data1',      // 'one' ← キー
    'two' => 'data2',      // 'two'
    'three' => 'data3'     // 'three'
);

// キーを指定し値の追加を行う
$array['four'] = 'data4';

// 追加されている
var_dump($array); // ['data1', 'data2', 'data3', 'data4', ]

var_dump($array['four']); // 'data4'
```

■ 配列と連想配列を使用した入れ子の配列

また、配列は入れ子にすることが可能で、配列、連想配列の両方が使えます。

```
// 配列の宣言
$array = array(
    'data1',
    'data2',
    'data3',
    // 連想配列の入れ子
    'array_in_array' => array(
        'data4',
        'data5',
        'data6'
    ),
    // 連想配列
    'data_7' => 'data7'
);

// データの取得
var_dump($array[2]);           // 'data3'
var_dump($array['array_in_array']); // ['data4','data5','data6']
var_dump($array['array_in_array'][0]); // 'data4'
var_dump($array['data_7']);    // 'data7'
```

NULL

NULLは特別な型です。

変数に**値が存在しないこと**を表します。

NULLは

- 変数にNULLが代入されている
- まだ値が何も代入されていない

場合にNULLと判定されます。

練習課題) 1. 型

*参照する ... `var_dump()` で値を参照すること。

1. 論理値

- (1) 値が `TRUE` の論理値型の変数を宣言し、参照してみましょう。
- (2) 値が `FALSE` の論理値型の変数を宣言し、参照してみましょう。

2. 整数

- (1) 適当な値の整数型の変数を宣言し、参照してみましょう。
- (2) $1 + 2$ の計算を行い、結果を参照してみましょう。
- (3) $50 - 35$ の計算を行い、結果を参照してみましょう。
- (4) 2×8 の計算を行い、結果を参照してみましょう。
- (5) $16 \div 4$ の計算を行い、結果を参照してみましょう。
- (6) 2 の2乗 の計算を行い、結果を参照してみましょう。
- (7) $100 \times ((50 - 39 + 70) \div 3)$ の計算を行い、結果を参照してみましょう。

3. 文字列

以下の**文字列型**の変数を宣言してください。

変数名	値
<code>\$hello</code>	'こんにちは'
<code>\$michael</code>	'マイケル'

- (1) 変数 `$hello` を参照してみましょう。
- (2) 変数 `$hello` と変数 `$michael` の文字列結合を行い、参照してみましょう。
- (3) 変数 `$hello` と文字列 'ボブ' の文字列結合を行い、参照してみましょう。

4. 配列

1. 配列

以下の値を格納する**配列型**の変数 **\$fruits** を宣言してください。

値	型
'バナナ'	文字列型
'りんご'	文字列型
'オレンジ'	文字列型

- (1) 変数 \$fruits を参照してみましょう。
- (2) 変数 \$fruits内よりキーが2である値を参照してみましょう。
- (3) 変数 \$fruits内よりキーを指定し、値 'りんご' を参照してみましょう。
- (4) 変数 \$fruitsに、値 'さくらんぼ' を追加し、参照してみましょう。

2. 連想配列

以下のキーと対応した値を格納する**連想配列型**の変数 **\$foods** を宣言してください。

キー	値	型
'野菜'	'キャベツ'	文字列型
'肉'	'牛肉'	文字列型

- (1) 変数 \$foods を参照してみましょう。
- (2) 変数 \$foods 内より、キーが '肉' の値を参照してみましょう。
- (3) 変数 \$foods に、キーが '主食'、値が '白米' を追加し、変数 \$foods を参照してみましょう。

■ 3. 配列と連想配列の組み合わせ

以下の値を格納する**配列型**の変数 **\$fruits** を宣言してください。

値	型
'バナナ'	文字列型
'りんご'	文字列型
'オレンジ'	文字列型

以下のキーと対応した値を格納する**連想配列型**の変数 **\$foods** を宣言してください。

キー	値	型
'野菜'	'キャベツ'	文字列型
'肉'	'牛肉'	文字列型

- (1) 変数 \$foods にキーが '果実', 値が \$fruits を追加し、変数 \$foods を参照してみましょう。
- (2) 変数 \$foods 内よりキーを指定し、値 'オレンジ' を参照してみましょう。

5. NULL

- (1) 値が**NULL**の、変数 \$null を宣言し、参照してみましょう。

※ 1. 型 は以上で完了です。

2. 制御文

次は値の判定や、処理を繰り返し行う書き方を学んでいきます。

制御文は名前のごとく処理の制御を行います。

処理と言うと難しく聞こえますが、値の宣言や計算、文字列結合も立派な処理です。

ゆっくりでいいので、やっていることを理解しながらやっていきましょう。

＊「[制御構造](#)」（[PHP公式言語リファレンス](#)）[🔗](#)

■ 例) 変数Aが1の時だけ処理を行う制御文

```
if ( $変数A == 1 ) {  
    // 変数Aが1の時のみ、この{}内の処理が走る  
}
```

分岐を行う場合には、**演算子を使用した式を条件として設定**していきます。

上記の例では

「変数Aが1の時」という**条件**を、

「\$変数A == 1」のような**式**に落とし込んでいますね。

この過程は条件分岐を行う上でとても大事なことです。

- まず、やりたいことを最小限の条件に切り分ける。
- 次に、それぞれの条件を式に落とし込む。
のような順序でやっていくと理解が進むと思います。

では、先にこの**式の書き方**を学んでおきましょう。

比較演算子

比較演算子は比較のための演算子です。

2つの値が等しいかどうかや、大きい小さいの判断に使用します。

これらの式は論理値型の値を返却します。

■ 比較演算子を使用した式

```
// 等しい 型の相互変換をした後で $a が $b に等しい時に TRUE。  
$a == $b
```

```
// 等しい $a が $b に等しく、および同じ型である場合に TRUE 。  
$a === $b
```

```
//等しくない 型の相互変換をした後で $a が $b に等しくない場合に TRUE。  
$a != $b
```

```
// 等しくない 型の相互変換をした後で $a が $b に等しくない場合に TRUE。  
$a <> $b
```

```
// 等しくない $a が $b と等しくないか、同じ型でない場合に TRUE 。  
$a !== $b
```

```
//より少ない $a が $b より少ない時に TRUE。  
$a < $b
```

```
// より多い $a が $b より多い時に TRUE。  
$a > $b
```

```
// より少ないか等しい $a が $b より少ないか等しい時に TRUE。  
$a <= $b
```

```
// より多いか等しい $a が $b より多いか等しい時に TRUE。  
$a >= $b
```

論理演算子

さらに、複雑な条件の指定が可能である**論理演算子**を学びます。
これらも比較演算子の組み合わせで**論理値型の値**を返却します。

■ 比較演算子と論理演算子を使用した式

```
// 論理積    $a および $b が共に TRUE の場合に TRUE  
$a and $b
```

```
//論理和 $a または $b のどちらかが TRUE の場合に TRUE  
$a or $b
```

```
// 排他的論理和    $a または $b のどちらかが TRUE でかつ両方とも TRUE でない場合  
に TRUE  
$a xor $b
```

```
//否定    $a が TRUE でない場合 TRUE  
! $a
```

```
// 論理積    $a および $b が共に TRUE の場合に TRUE  
$a && $b
```

```
//論理和 $a または $b のどちらかが TRUE の場合に TRUE  
$a || $b
```

条件分岐

始めに**条件分岐**を学習していきましょう。

特定の場合にのみ処理を行いたかったり、**値によって行う処理を分けたい**場合に、条件分岐を使用します。条件分岐には以下の**2つ**の種類があります。

- **if**
- **switch**

条件分岐は処理の基本です。

難しいものではありませんので、しっかりと覚えておきましょう。

if文

if文はもっともシンプルな分岐です。

特定の場合にのみ処理を行うことができます。

■ if文を使用した条件分岐

if文は以下のように書きます。

```
if (式) {  
    //式がTRUEの場合の処理  
}
```

■ if文を使用した条件分岐の例

if文は、まず**式**を評価します。

この値が**TRUE**の場合のみ**処理が実行されます**。

FALSEの場合には、**処理の実行は行われません**。

例として**比較演算子**を使用し、**数字A,数字Bの値が同じ場合**のみ、文字列に値を入れてみましょう。

```
$number_A = 1;  
$number_B = 1;  
$word = '';  
  
if ($number_A === $number_B) {  
    $word = '2つの数字は同じです';  
}  
  
var_dump($word ); // 2つの数字は同じです  
  
// $number_A と $number_Bがの値が一致しない時には、''が入っている。
```

■ 条件式に値を設定する

式には**論理値を返す値**を設定することができます。

また、変数のみを設定した場合には、**暗黙の型変換**が行われます。

試しにいろんな型の変数を作成し、式に設定してみましょう。

```
// 整数型での例

if (1) {
    // 出力される
    print 'TRUE!';
}

if (') {
    // 出力されない
    print 'TRUE!';
}
```

比較の際に注意すべき点

暗黙の型変換について注意すべき点があります。

■ 暗黙の型変換

下記の条件分岐ですがどうなるでしょうか？

```
if ( '' == 0 ) {  
    print '値は同じです';  
}
```

これはなんと、分岐に入ってしまうのです。

結果

値は同じです

0, "", null などの値は、暗黙の型変換時に **FALSE** として扱われます。

さらにこの比較では、比較演算子の **==** を使用しているので、**値のみの比較** となり、値は同じと判断されてしまうのです。

詳しくは、[型の相互変換](#) と [比較について](#) を参照してください。

これを避けるためには、**値と型の比較** を行う **===** を使用しましょう。

```
if ( '' === 0 ) {  
    print '値は同じです';  
}
```

結果

else (if文)

elseとは英語で「**それ以外の**」を意味します。

その名の通り、if文で指定した「**特定の場合**」**以外の場合**の分岐を行うのが**else**です。

elseではif文の式が**TRUEにならなかった場合**処理を行います。

■ elseを含むif文を使用した条件分岐

elseを含むif文は、以下のように書きます。

```
if ( 式 ) {  
    //式がTRUEの場合の処理  
} else {  
    //式がFALSEの場合の処理  
}
```

■ elseを含むif文を使用した条件分岐の例

if文の例に記述を加えて

- **数字A,数字Bの値が同じ場合**のみ、文字列に「2つの数字は同じです」を入れる。
- **それ以外の時**には文字列に「2つの数字は違います」を入れる。

2つの条件で処理を書いてみましょう。

```
$number_A = 1;  
$number_B = 2;  
$word = '';  
  
if ($number_A === $number_B) {  
    $word = '2つの数字は同じです';  
} else {  
    // $number_A と $number_B の値が一致しないのでelseの処理が実行される。  
    $word = '2つの数字は違います';  
}  
  
var_dump($word) ; // 2つの数字は違います
```


elseif, else if (if文)

ここまでで、if文とelseを使用した

- 特定の場合のみ処理を実行する。
- それ以外の場合に処理を実行する。

はできるようになったと思います。

では、さらに**複数の特定条件を指定できる、elseif**を学んでいきましょう。

■ elseifを含むif文を使用した条件分岐

elseifを含むif文は、以下のように書きます。

```
if ( 式1 ) {  
    // 式1がTRUEの場合の処理  
} elseif ( 式2 ) {  
    // 式2がTRUEの場合の処理  
}
```

■ else, elseifを含むif文を使用した条件分岐の例

では次に、else, elseif, を使用して

- 数字A, 数字Bの値が同じ場合のみ、文字列に「2つの数字は同じです」を入れる。
- **数字Aが1の場合には、文字列に「数字Aは1です、Bは違います」を入れる。**
- それ以外の時には文字列に「2つの数字は違います」を入れる。

3つの条件で処理を書いてみましょう。

```
$number_A = 1;  
$number_B = 2;  
$word = '';  
  
if ($number_A === $number_B) {  
    $word = '2つの数字は同じです';  
} elseif ($number_A === 1) {  
    // $number_Aが1なのでこの処理に入る  
    $word = '数字Aは1です、Bは違います';  
} else {  
    $word = '2つの数字は違います';  
}  
  
var_dump($word) ; // 数字Aは1です、Bは違います
```

if文についての補足

if文の入れ子

if文は以下のように入れ子にすることができます。

```
$number_A = 1;
$number_B = 1;
$word = '';

if ($number_A === $number_B) {
    // 数字が同じ場合
    $word = '数字は同じ値です';

    if ($number_A === 1) {
        // 数字が同じ場合 かつ $number_Aの値が1
        $word = '数字は同じ値 かつ 値は1です';
    }
}

var_dump($word) ; //数字は同じ値 かつ 値は1です
```

論理演算子を使用した条件の書き換え

以下の条件分岐は同じ処理になります。

```
$number_A = 1;
$number_B = 1;
$word_1 = '';
$word_2 = '';

// 比較演算子のみで条件分岐を行う場合
if ($number_A === $number_B) {
    // 数字が同じ場合
    if ($number_A === 1) {
        // $number_Aの値が1
        $word_1 = '数字は同じ値 かつ 値は1です';
    }
}

// 論理演算子を使用して条件分岐を行う場合
if ($number_A === $number_B and $number_A === 1) {
    // 数字が同じ場合 かつ $number_Aの値が1
    $word_2 = '数字は同じ値 かつ 値は1です';
}

var_dump($word_1) ; //数字は同じ値 かつ 値は1です
var_dump($word_2) ; //数字は同じ値 かつ 値は1です
```

switch文

次は**switch文**です。

switch文はif文と同じ、条件分岐文です。

switch文は、**評価したい式の返却値**によって処理を分けることができます。

■ switch文を使用を使用した条件分岐

case 式の返却値 で分岐条件を記述し、その場合の処理を書いて行きます。

breakは**処理終了の意**です。忘れないように記述しましょう。

```
switch (式) {  
    case 値1 :  
        // 式の返却値が値1 の場合の処理  
        break;  
    case 値2 :  
        // 式の返却値が値2 の場合の処理  
        break;  
    case 値3 :  
        // 式の返却値が値3 の場合の処理  
        break;  
}
```

■ switch文を使用した条件分岐の例

例として変数 \$val の値で分岐してみましょう

```
$val = 1  
  
switch ($val) {  
    case 1 :  
        //この処理に入る  
        print '$valの値は1 です';  
        break;  
    case 2 :  
        print '$valの値は2 です';  
        break;  
    case 3 :  
        print '$valの値は3 です';  
        break;  
}
```

結果

\$valの値は1 です

defaultの設定(switch文)

また、switch文では、**いずれの条件にも当てはまらない場合**の処理を設定することが可能です。

default はcaseと同じように記述します。

```
$val = 4

switch ($val) {
    case 1:
        print '$valの値は1です';
        break;
    case 2:
        print '$valの値は2です';
        break;
    case 3:
        print '$valの値は3です';
        break;
    default:
        //この処理に入る
        print '$valの値は1,2,3以外の値です';
}
```

結果

\$valの値は1,2,3以外の値です

ループ

続いては**ループ**です。

ループでは**処理を繰り返す**ことができます。

■ 例) 文字列 'テスト表示' を5回表示する

例えば、5回同じ処理を繰り返したい時などは、

```
print 'テスト表示'; // 1
print 'テスト表示'; // 2
print 'テスト表示'; // 3
print 'テスト表示'; // 4
print 'テスト表示'; // 5
```

上記のように5回同じ処理を書くことで実現できますが、これが100回となってくると大変ですよ。

そこでループ文の出番です。

上記の処理と同じことを以下の書き方で実現することができます。

■ 例) ループを使用して、文字列 'テスト表示' を5回表示する

```
for ($i=1; $i<=5; $i++){
    print 'テスト表示';
}
```

短くなりましたね！素敵です。

この書き方ならば、5回でも100回でも少ない行数で実現できるんです。

■ ループの種類

ループでは以下の3つの種類のループ文を学んでいきます。

基本制御の

- **for**
- **while**

配列型の値を使用する際に有用な

- **foreach**

加算子/減算子

ループの基本文法に入る前に簡単な演算子を学びます。

加算子は数値に1を加える演算子です。

減算子は数値から1を引く演算子です。

前置と**後置**の違いは値を返すタイミングが違います。

```
// 前置加算子 $a に 1 を加え、$a を返します。
++$a

// 後置加算子 $a を返し、$a に1を加えます。
$a++

// 前置減算子 $a から 1 を引き、$a を返します。
--$a

// 後置減算子 $a を返し、$a から 1 を引きます。
$a--
```

■ 加算子/減算子 を使用した加減の例

```
// 後置加算
$a = 5;
var_dump( $a++ ); // int(5)
var_dump( $a );   // int(6)

// 前置加算
$a = 5;
var_dump( ++$a ); // int(6)
var_dump( $a );   // int(6)

// 後置減算
$a = 5;
var_dump( $a-- ); // int(5)
var_dump( $a );   // int(4)

// 前置減算
$a = 5;
var_dump( --$a ); // int(4)
var_dump( $a );   // int(4)
```

for

for文は基本的なループ制御文です。

まず、書き方をみてみましょう。

■ forを使用したループ処理

```
for (初期値式; 継続条件式; 増減式){  
    // 処理  
}
```

for文では、ループの回数を変数（**インデックス**）を使用して制御します。

出てきた以下の3つの式では、インデックスの設定を行います。

- **初期値式** . . . ループ前に一回実行される。インデックスの初期値を設定する。
- **継続条件式** . . . ループ毎に実行される。式がTRUEの時のみ、次のループが実行される。
- **増減式** . . . ループ毎に実行される。インデックスの増減を設定する。

■ インデックス設定の例

冒頭のループを例に式設定の方法をみてみましょう。

```
// 例) ループを使用して、文字列 'テスト表示' を 5 回表示する  
  
for ($i=1; $i<=5; $i++){  
    print 'テスト表示';  
}
```

5回処理を実行したい時の設定は、

<code>\$i = 0</code>	// 初期値式	→ <code>\$i</code> に 0 を設定 (ループ開始前に一回のみ実行)
<code>\$i < 5</code>	// 継続条件式	→ <code>\$i</code> が 5 以下である時 (ループ毎にかくにん)
<code>\$i++</code>	// 増減式	→ <code>\$i</code> に 1 を足す

このようになっています。

次に、この時の実際のループの動きをみてみましょう。

■ ループ時の動き

理解しやすいように、少しコードに手を加えてみます。

```
for ($i=1; $i<=5; $i++){
    var_dump($i) ; // 変数$iの中身を確認する。追加！
    print 'テスト表示';
    print '<br/>'; // 改行する。追加！
}
```

ループ毎に変数*\$i*の中身を確認するコードと、改行を追加しました。
以下が結果画面に表示されているはずです。みてみましょう。

結果

```
int(1) テスト表示
int(2) テスト表示
int(3) テスト表示
int(4) テスト表示
int(5) テスト表示
```

各行の `int()` は `var_dump($i)` の結果です。

初期値式 (`$i=1;`) で設定したので、1行目では **int(1)** となっています。

1回目の処理が終わったのちに**継続条件式 (`$i<=5;`)** は **TRUE**を返すので次のループに入ります。

2行目では1増えて、**int(2)** となっていますね、これは **増減式 (`$i++;`)** が動いているためです。

このような流れで、*\$i*を1増やししながら、処理を繰り返していきます。

そして、*\$i* が6になった時に**継続条件式 (`$i<=5;`)** が **FALSE**になるので、処理が終了します。

■ 無限ループに注意

継続条件式をうまく設定しないとループが終了しない**無限ループ**になってしまうことがあります。
注意しましょう。

```
for ($i=1; $i=5; $i++){
    // 継続条件式が $i=5; になっている
    // 意図したのは $i==5;
    print 'テスト表示';
}
```

結果

```
テスト表示テスト表示テスト表示テスト表示テスト表示テスト表示テスト表示テスト表示テスト表示...

```

while

続いては **while文**です。

記述方法は違いますが、やっていることはfor文と変わりありません。

■ whileを使用したループ処理

whileが扱うのは**継続条件式のみ**です。

初期値式 や、**増減式**は自分で記述しなければいけません。

```
while ( 継続条件式 ){  
    // 処理  
}
```

■ whileを使用したループ処理の例

for文の例で使った、「ループを使用して、文字列 'テスト表示' を 5 回表示する」がありましたね。

```
// for文の例  
for ($i=1; $i<=5; $i++){  
    print 'テスト表示';  
}
```

これをwhile文を使用して実装してみます。

```
$i=1; // インデックスの初期値を設定。（初期値式）  
  
while ( $i<=5 ){  
    print 'テスト表示';  
    $i++; // インデックスを1増やす。（増減式）  
}
```

こうなります。

for文が理解できていればわかりやすいと思います。

増減式の記述を忘れると無限ループになってしまうので気をつけましょう。

forとwhileの違いは？

forとwhileはできることは同じなので、違いはほぼありません。
ですが、

- 繰り返す回数が決まっている場合にはforを
- 特定の条件で繰り返しを終了したい場合にはwhileを

使用場合があります。

理由としてはwhile文において、継続条件のみの記述で済むからです。

ですがこれは決して、**決まりではない**ので「好きな方を使う」くらいの気持ちで良いと思います。

例) 30回繰り返す処理

```
// for文を使用する
for ( $i=1; $i<=30; $i++){
    // 30回繰り返す処理
}
```

例) 特定の条件で繰り返しを終了する。

```
$stop = false;

while ( $stop == true ){
    // どこかのタイミングで $stop = true になるような処理
}
```

foreach

foreachは配列の扱いに長けたループ文です。

1 ループごとに配列の要素を1つずつ参照することができます。

また、**配列**と**連想配列**で記述が変わってきます。

■ foreachを使用した配列の操作

配列を扱う場合、要素1つの**値の参照**が可能です。

ループの回数は\$array(配列)の**要素の数の分**ループします。

```
foreach ( $配列 as $値代入変数 ){  
    // $値代入変数 : 要素の1つが代入される  
}
```

■ foreachを使用した配列の操作の例

例として、配列の要素を1つずつ表示してみましょう。

```
$data_array = array( 'data1', 'data2', 'data3', 'data4', 'data5' );  
  
foreach ( $data_array as $data){  
    print $data; // 要素1つの表示  
    print '<br/>'; // 改行  
}
```

結果

```
data1  
data2  
data3  
data4  
data5
```

■ ループ内の動き

1 ループごとに\$data_arrayの要素が\$dataに代入されます。

例でみると、

1 ループ目には、'data1' が、

2 ループ目には、次の'data2' が、といったように\$dataの値が変わっていることがわかります。

■ foreachを使用した連想配列の操作

配列の場合には、値のみの参照でした。

ですが、**連想配列を扱う場合にはキーと値**の両方の参照が可能です。

```
foreach ( $連想配列 as $キー代入変数 => $値代入変数 ){
    // $キー代入変数      : 要素1つのキーが代入される
    // $値代入変数        : 要素1つの値が代入される
}
```

■ foreachを使用した連想配列の操作の例

各要素のキーと値を「キー：値」の形で表示してみましょう。

```
$data_array = array(
    'list_1' => 'data1',
    'list_2' => 'data2',
    'list_3' => 'data3',
    'list_4' => 'data4',
    'list_5' => 'data5'
);

foreach ( $data_array as $key => $val ){
    print $key.' : '.$val; // 要素1つの 「キー : 値」を表示
    print '<br/>'; // 改行
}
```

結果

ループごとにキーと値が取り出せていることが分かりますね。

```
list_1 : data1
list_2 : data2
list_3 : data3
list_4 : data4
list_5 : data5
```

練習課題) 2. 制御文

現在作成中です ...

1. 条件分岐

■ if文

■ switch文

2. ループ

■ for文

■ while文

■ foreach文

3. 発展問題

＊ 2. 制御文 は以上で完了です。

3. 関数

続いては関数です。

■ 同じ処理の繰り返しの排除

例えば、このような処理があったとしましょう。

エラーを教えてくれる処理

```
$val_1 = true ;
$val_2 = true ;
$val_3 = true ;

if($val_1){
    print '大変！';
    print '<br/>';
    print 'エラーです！';
    print '<br/>';
}
if($val_2){
    print '大変！';
    print '<br/>';
    print 'エラーです！';
    print '<br/>';
}
if($val_3){
    print '大変！';
    print '<br/>';
    print 'エラーです！';
    print '<br/>';
}
```

この処理、なんだかもう少し綺麗に書ける気がしませんか？

同じようなことを繰り返していますよね？

もし、表示する文言を変えたい場合には3箇所を変えなければなりません。

■ 関数を使うと

これから学ぶ関数を使うと、**処理をまとめておいて、呼び出す**ことができるようになります。

また、処理を関数にすることを**関数化**と言います。

関数化ができるようになると、

- コードの圧縮（行が少なくなる）
- 処理の使い回しができる

ようになります。早速、学んでいきましょう。

関数

関数の宣言

関数は **function()** を使用して宣言します。

```
function 関数名(){  
    // 処理  
}
```

関数の呼び出し

関数の呼び出し方は以下です。

```
function 関数名();
```

また、必ず、関数を宣言した後に関数の呼び出しを行ってください。
宣言されていない状態で関数の呼び出しを行うとエラーになります。

関数化の例

早速、冒頭例のエラー表示の部分を関数化してみましょう。

print_errors()の宣言

```
function print_errors(){  
    print '大変!';  
    print '<br/>';  
    print 'エラーです!';  
    print '<br/>';  
}
```

次に、エラー表示の処理の関数 **print_errors()** を呼び出してみましょう。

print_errors()の呼び出し

```
print_errors();
```

結果

```
大変!  
エラーです!
```

うまく動いているようですね！

■ 関数に値を渡したい

さて、関数を使い、冒頭の例の処理が短くなりました。
見やすいですし、文言の変更も1箇所済みそうです。

エラーを教えてくれる処理（関数化1）

```
$val_1 = true ;  
$val_2 = true ;  
$val_3 = true ;  
  
function print_errors(){  
    print '大変！';  
    print '<br/>';  
    print 'エラーです！';  
    print '<br/>';  
}  
  
if($val_1){  
    print_errors();  
}  
if($val_2){  
    print_errors();  
}  
if($val_3){  
    print_errors();  
}
```

ですがまだ繰り返しがありますね。

```
if($val_1){  
    print_errors();  
}  
if($val_2){  
    print_errors();  
}  
if($val_3){  
    print_errors();  
}
```

関数にしたいのですが、各if文で違う変数を見えています。
今学んだものだけでは、関数化が難しそうですね。

しかし、なんと、**関数に値を渡す**こともできるのです。
早速やってみましょう。

引数のある関数

■ 引数のある関数の宣言

関数に値を渡したい時には**関数の宣言時に引数**を設定します。

引き数は**関数のスコープ内でのみ有効**です。

また、**引数はいくつでも設定**することができます。

```
function 関数名($引数1, $引数2, $引数3, ...){  
    // $引数nを使った処理  
}
```

■ 引数のある関数の呼び出し

引数のある関数の呼び出し時には引数に使用したい値を設定します。

また、**引数の数があっていないとエラーになる**ので注意しましょう。

```
関数名($引数1, $引数2, $引数3, ...);
```

■ 関数内の動き

宣言時の引数それぞれに、呼び出し時に設定された引数が代入されます。

```
// 関数の宣言  
function print_arg($arg){  
    print $arg;  
}  
  
// 変数 $test_argument の宣言  
$test_arg_a = 'テスト用の引数aです';  
$test_arg_b = 'テスト用の引数bです';  
  
// 引数に設定し、関数を呼び出す  
print_arg($test_arg_a); // $argに$test_arg_aが代入される  
print_arg($test_arg_b); // $argに$test_arg_bが代入される
```

結果

```
テスト用の引数aです  
テスト用の引数bです
```

■ 関数の力

さあ、今度は引数のある関数を使用して冒頭例の書き換えを行ってみます。

エラーを教えてくれる処理（関数化2）

```
$val_1 = true ;
$val_2 = true ;
$val_3 = true ;

function is_error($value){
    if($value){
        print '大変！';
        print '<br/>';
        print 'エラーです！';
        print '<br/>';
    }
}

is_error($val_1);
is_error($val_2);
is_error($val_3);

}
```

かなり簡潔になりました。

関数を使えば、どんなに複雑な処理でも一行で呼び出すことができますね。これが関数の力です。

基本的に、2箇所以上で同じ処理を行う場合には関数化を考えてみましょう。

値を返す関数

関数の基本がわかったところで少し発展してみましょう。
まず以下のソースをみてみてください。

```
$val_1 = 3 ;  
$val_2 = 5 ;  
$val_3 = 17 ;  
  
$val_1 = $val_1*2 ;  
$val_2 = $val_2*2 ;  
$val_3 = $val_3*2 ;
```

今度の変数の値を2倍するという処理になります。これらの処理を関数化してみましょう。

■ returnで値を返す

関数内で **return** を使うと値を返却することができます。
今回は**受けとった値を2倍して返してみよう**。

■ 返却値のある関数の使い方

```
// 関数の宣言  
function double($val){  
    $double_val = $val*2; // 値を2倍する  
    return $double_val ; // 値を返す  
}  
  
$val_1 = 3 ;  
  
// 関数の呼び出し  
$val_1 = double($val_1);  
  
var_dump($val_1) // int(6)
```

＊ 3. 関数 は以上で完了です。

ちょっと休憩 ...

4. クラス

次は**クラス**です。

クラスは**オブジェクト指向**を実現するための文法です。

オブジェクト指向は、**設計思想**の一つで、**モノとして考える**特徴があります。

設計思想とは、作りたいものを作るときにどうやって作っていくかの考えのことです。

設計について

オブジェクト指向の概要とメリット

オブジェクト指向を学ぶにあたって、先に**概要**と**メリット**のお話をしましょう。
まず、オブジェクト指向とは概念であり、実装の方法や考え方のひとつです。

