

# Методические указания по теме “Осваиваем работу с функциями”

[Основы работы с функциями](#)

[Параметры функции](#)

[Передача параметров по значению и ссылке](#)

[Необязательные параметры](#)

[Переменное количество параметров](#)

[Глобальные переменные](#)

[Статические переменные](#)

[Возврат массива функцией](#)

[Рекурсивные функции](#)

[Вложенные функции](#)

[Динамическое имя функции](#)

[Анонимные функции](#)

[Проверка существования функции](#)

[Вспомогательные функции](#)



# Основы работы с функциями

**Функция** - поименованный фрагмент программного кода (подпрограмма), к которому можно обратиться из другого места программы. С именем функции неразрывно связан адрес первой инструкции (оператора), входящей в функцию, которой передаётся управление при обращении к функции. После выполнения функции, управление возвращается обратно в адрес возврата — точку программы, где данная функция была вызвана.

Функция объявляется при помощи ключевого слова *function*, после которого следует имя функции, в круглых скобках параметры функции и в фигурных скобках записываются различные операторы, составляющие тело функции:

```
function MyFunction()
{
  // операторы
}
```

Если функция возвращает какое-либо значение, в теле функции обязательно должен присутствовать оператор `return`:

```
function MyFunction()
{
  // Вычисления
  return $ret; // возвращается значение переменной $ret
}
```

## Простой пример работы с функциями:

```
<?php
function get_sum()
{
  $sum = 10 + 5;
  return $sum;
}
echo get_sum(); // выводит 15
?>
```

Эта функция не принимает ни одного аргумента, а просто вычисляет сумму и возвращает полученный результат. После этого она вызывается в теле оператора *echo* для вывода результата в браузер. Модифицируем эту функцию так, чтобы она не возвращала полученный результат, а выводила его в браузер. Для этого достаточно внести оператор *echo* в тело функции.

```
<?php
function get_sum()
{
  $sum = 10 + 5;
  echo $sum;
}
```

```
}  
get_sum();  
?>
```

**Во многих языках программирования функция не может вызываться до ее объявления. В PHP отсутствуют подобные ограничения, функция может вызываться до ее объявления.**

Это правило изменяется, если объявление функции осуществляется внутри фигурных скобок. Функции могут быть объявлены в блоке, обрамленном фигурными скобками.

```
<?php  
// Объявляем логическую переменную  
$flag = TRUE;  
  
// Если переменная $flag равна TRUE, объявляем функцию  
if ($flag)  
{  
    function get_sum()  
    {  
        $sum = 10 + 5;  
        echo $sum;  
    }  
}  
  
// Вызываем функцию, если переменная $flag равна TRUE  
if ($flag) get_sum(); // 15  
?>
```

Однако объявить функцию позднее ее вывода в этом случае уже не получится.

```
<?php  
// Объявляем логическую переменную  
$flag = TRUE;  
  
// Вызываем функцию, если переменная $flag равна TRUE  
if ($flag) get_sum(); // Ошибка  
  
// Если переменная $flag равна TRUE, объявляем функцию  
if ($flag)  
{  
    function get_sum()  
    {  
        $sum = 10 + 5;  
        echo $sum;  
    }  
}  
?>
```

Попытка вызова функции, объявленной условно, раньше объявления приводит к генерации ошибки.

**!** Fatal error: Call to undefined function get\_sum() in C:\OpenServer\domains\basic-project.local\example-1.php on line 6

**Call Stack**

#	Time	Memory	Function	Location
1	0.0010	121960	{main}()	..\example-1.php:0

**Dump \$ \_SERVER**

<code>\$_SERVER['REMOTE_ADDR']</code>	= string '127.0.0.1' (length=9)
<code>\$_SERVER['REQUEST_METHOD']</code>	= string 'GET' (length=3)

**Dump \$ \_SESSION**

<code>\$_SESSION['*']</code>	= <i>undefined</i>
------------------------------	--------------------

**Dump \$ \_REQUEST**



## Параметры функции

Можно значительно увеличить гибкость функции, если складываемые числа будут передаваться в качестве параметров.

```
<?php
function get_sum($left, $right)
{
    $sum = $left + $right;
    echo $sum;
}
get_sum(10, 5); // 15
?>
```

Переменная, содержащая значение, переданное через аргумент, называется параметром функции, т. е. в примере числа 10 и 5 являются аргументами, а переменные \$left и \$right — параметрами.

В качестве параметров могут выступать выражения, и даже другие функции. В PHP выражения в этом случае вычисляются слева направо

```
<?php
function funct($left, $middle, $right)
{
    echo $left . "<br>";
    echo $middle . "<br>";
    echo $right . "<br>";
}
$i = 10;
funct(++$i, $i = $i * 2, --$i);
?>
```

Результатом выполнения скрипта из будет такая последовательность чисел:

```
11
22
21
```



## Передача параметров по значению и ссылке

В рассмотренных примерах аргументы функции передаются по значению, т. е. значения параметров изменяются только внутри функции, и эти изменения не влияют на значения переменных за пределами функции.

```
<?php
function get_sum($var) // аргумент передается по значению
{
    $var = $var + 5;
    return $var;
}
$new_var = 20;
echo get_sum($new_var); // 25
echo "<br>$new_var"; // 20
?>
```

Для того чтобы переменные, переданные функции, сохраняли свое значение при выходе из нее, применяется передача параметров по ссылке. Для этого перед именем переменной необходимо поместить амперсанд (&):

```
function get_sum(&$var)
```

В этом случае переменная *\$var* будет передана по ссылке. В случае, если аргумент передается по ссылке, при любом изменении значения параметра происходит изменение переменной-аргумента

```
<?php
function get_sum(&$var) // аргумент передается по ссылке
{
    $var = $var + 5;
    return $var;
}
$new_var = 20;
echo get_sum($new_var); // выводит 25
echo "<br>$new_var"; // выводит 25
?>
```



## Необязательные параметры

Параметры можно объявлять как необязательные. Для этого при объявлении параметра необходимо присвоить ему значение по умолчанию.

```
<?php
function get_sum($left = 10, $right = 5)
{
    $sum = $left + $right;
    echo $sum;
}
get_sum(); // выводит 15
get_sum(5); // выводит 10
get_sum(5, 0); // выводит 5
?>
```

Если функции `get_sum()` не передаются параметры, она успешно производит вычисления с участием параметров по умолчанию. Если функция содержит множество обязательных и необязательных параметров, то все обязательные параметры следует располагать до необязательных. В примере ниже приводится некорректное объявление функции `get_sum()`, в котором необязательный параметр предшествует обязательному.

```
<?php
function get_sum($left = 10, $right)
{
    $sum = $left + $right;
    echo $sum;
}
get_sum(5);
?>
```



## Переменное количество параметров

C-подобные языки программирования обычно поддерживают синтаксис функций с переменным количеством параметров. В PHP для этого необходимо объявить функцию без параметров — такой функции можно передавать любое их количество, без каких-либо последствий.

```
<?php
function get_sum() {
    echo "Вызов функции";
}
get_sum(5, "Второй параметр", 124);
?>
```

Можно убедиться, что интерпретатор PHP не генерирует ни замечаний, ни предупреждений (в противовес ситуации, когда функция имеет хотя бы один параметр). Для работы с переменным количеством параметров в PHP предусмотрены специальные функции.

Функция	Описание
func_num_args()	Возвращает количество параметров, переданных функции
func_get_args()	Возвращает массив с параметрами, переданными функции
func_get_arg(\$arg_num)	Возвращает значение параметра с номером \$arg_num. Если параметр с таким номером не существует, функция возвращает FALSE

Ниже приводится пример функции, которая выводит значения всех своих параметров.

```
<?php
// Объявляем функцию
function get_parameters()
{
    for ($i = 0; $i < func_num_args(); $i++)
    {
        echo "Параметр номер $i: ".func_get_arg($i)."<br>";
    }
}

// Вызываем функцию
echo get_parameters("Hello", "world", "!");
?>
```

Результатом выполнения скрипта будут следующие строки:

```
Параметр номер 0: Hello,
Параметр номер 1: world
Параметр номер 2: !
```

## Глобальные переменные

Переменные в функциях имеют локальную область видимости. Это означает, что даже если локальная (внутри функции) и внешняя (вне функции) переменные имеют одинаковые имена, то изменение локальной переменной никак не повлияет на внешнюю переменную.

```
<?php
function get_sum()
{
    $var = 5;    // локальная переменная
    echo $var;
}
$var = 10;    // внешняя переменная
get_sum();    // выводит 5 (локальная переменная)
echo "<br>$var"; // выводит 10 (внешняя переменная)
?>
```

Локальную переменную можно сделать внешней, если перед ее именем указать ключевое слово `global`. В этом случае изменения как внутри функции, так и вне ее будут влиять на переменную, а сама переменная будет называться глобальной. Если локальная переменная объявлена как `global`, то к ней возможен доступ из любой части программы.

```
<?php
function get_sum()
{
    global $var;
    $var = 5; // изменяем глобальную переменную
    echo $var;
}
$var = 10;
echo "$var<br>"; // выводит 10
get_sum();    // выводит 5(глобальная переменная изменена)
echo "$var<br>"; // выводит 5
?>
```

Доступ к глобальным переменным можно получить также через суперглобальный массив `$GLOBALS`.

```
<?php
function get_sum()
{
    $GLOBALS['var'] = 20; // изменяем глобальную переменную $var
    echo($GLOBALS['var']);
}
$var = 10;
echo "$var<br>"; // выводит 10
get_sum();    // выводит 20 (глобальная переменная изменена)
?>
```

Массив \$GLOBALS доступен в области видимости любой функции и содержит все глобальные переменные, которые используются в программе.

## Статические переменные

Временем жизни переменной называется интервал выполнения программы, в течение которого она существует. Поскольку локальные переменные имеют своей областью видимости функцию, то время жизни локальной переменной определяется временем выполнения функции, в которой она объявлена. Это означает, что в разных функциях совершенно независимо друг от друга могут использоваться переменные с одинаковыми именами. Локальная переменная при каждом вызове функции инициализируется заново, поэтому функция-счетчик всегда будет возвращать значение 1.

```
<?php
function counter()
{
    $counter = 0;
    return ++$counter;
}
?>
```

Для того чтобы локальная переменная сохраняла свое предыдущее значение при новых вызовах функции, ее можно объявить статической при помощи ключевого слова `static`.

```
<?php
function counter()
{
    static $counter = 0;
    return ++$counter;
}
?>
```

В скрипте `$counter` устанавливается в ноль при первом вызове функции и при последующих вызовах функции помнит, каким было значение переменной при предыдущих вызовах.

Временем жизни статических и глобальных переменных является время выполнения сценария. То есть если пользователь перезагружает страницу, что приводит к новому выполнению сценария, переменная `$counter` инициализируется заново.

## Возврат массива функцией

Функция может возвращать массив в качестве значения, для этого достаточно передать его в качестве параметра оператору `return`. Более того, такой массив может создаваться динамически при помощи конструкции `array()`. К такому приему прибегают всякий раз, когда функция должна вернуть несколько значений, а передача значений по ссылке не допускается.

Ниже демонстрируется функция `format_size()`, которая принимает в качестве значения размер файла в байтах и возвращает массив, первый элемент которого содержит размер в байтах, второй — в килобайтах, третий — в мегабайтах, а четвертый — в гигабайтах.

```
<?php
function format_size($byte)
{
    $kbyte = $byte / 1024;
    $mbyte = $kbyte / 1024;
    $gbyte = $mbyte / 1024;

    return array($byte, $kbyte, $mbyte, $gbyte);
}
?>
```

Оперировать массивом в скрипте не всегда удобно, особенно, если он имеет постоянное небольшое количество элементов. Поэтому часто при вызове функции массив сразу же разбивается на переменные при помощи конструкции `list()`.

```
<?php
...
list($byte, $kbyte, $mbyte, $gbyte) = format_size(18642678);
?>
```



# Рекурсивные функции

Рекурсия — это вызов функцией самой себя, пример рекурсивной функции приведен ниже

```
<?php
function callself($counter)
{
    // Если параметр $counter больше, продолжаем рекурсивный спуск
    if ($counter>0)
    {
        // Уменьшаем значение параметра $counter и выводим его значение
        // в окно браузера
        echo($counter--)."<br>";
        // Осуществляем рекурсивный вызов функции callself()
        callself($counter);
    }
    // Если значение параметра меньше или равно 0, прекращаем
    // рекурсивный спуск
    else return;
}
// Вызываем функцию callself()
callself(4);
?>
```

Результатом работы функции будет последовательность цифр:

```
4
3
2
1
```

Функция `callself()` вызывает саму себя до тех пор, пока ее параметр `$counter` положителен и не равен нулю. Рекурсивных функций по возможности стараются избегать, т.к. они относятся к трудным по восприятию конструкциям языка, и отладка их достаточно сложна, особенно, когда приходится иметь дело не с простейшей рекурсивной функцией, представленной в выше, а со сложной функцией, осуществляющей рекурсивный вызов в нескольких местах функции.

Опасность использования неотлаженных рекурсивных функций заключается в возможности перехода их в режим бесконечной рекурсии, когда условие, прекращающее спуск вниз по рекурсии из-за ошибки, не наступает, в результате чего, как и в случае бесконечных циклов, наступает зависание программы.

Практически в любом случае можно избежать рекурсивных функций. Исключение составляют задачи, так или иначе связанные с обходом деревьев. К таким задачам относится, например, удаление каталогов, когда число файлов и подкаталогов заранее не известно и необходимо вызывать функцию удаления до тех пор, пока не будут удалены файлы на самом глубоком уровне вложенности.

## Вложенные функции

Язык программирования PHP позволяет объявлять функции внутри другой функции. В отличие от обычных функций, вложенная функция не может использоваться до тех пор, пока не будет осуществлен вызов основной функции, который произведет объявление вложенной.

```
<?php
// Объявление внешней и вложенной функций
function outter()
{
    function inner()
    {
        echo "Hello, world!";
    }
}

// Вызываем функцию outter(), чтобы объявить функцию inner()
outter();

// Функция inner() не может быть вызвана до тех пор,
// пока не будет вызвана функция outter();
inner();
?>
```



## Динамическое имя функции

По аналогии с переменными, имя функции может быть динамическим и храниться в строковой переменной — передача такой переменной оператором круглых скобок (с параметрами, если они требуются) приводит к вызову функции.

### Например:

```
<?php
// Объявление функций
function hello()
{
    echo "Hello!";
}
function bye()
{
    echo "Bye!";
}

// Случайный выбор функции
if (rand(0, 1)) $var = "hello";
else $var = "bye";

// Вызов функции
$var();
?>
```



## Анонимные функции

Анонимные функции, также известные как замыкания (*closures*), позволяют создавать функции, не имеющие определенных имен. Они наиболее полезны в качестве значений callback-параметров, но также могут иметь и множество других применений.

### Пример

```
<?php
$greet = function($name)
{
    printf("Hello %s<br />", $name);
};

$greet('World');
$greet('PHP');
```

Замыкания могут также наследовать переменные из родительской области видимости. Любая подобная переменная должна быть передана в языковую конструкцию *use*.

### Пример

```
<?php
$message = 'hello';

// Без "use"
$example = function () {
    var_dump($message);
};
echo $example();

// Наследование $message
$example = function () use ($message) {
    var_dump($message);
};
echo $example();

// Наследуемое значение переменной определяется в момент создания функции,
// а не в момент ее вызова
$message = 'world';
echo $example();

// Сброс $message
$message = 'hello';

// Наследование по ссылке
$example = function () use (&$message) {
    var_dump($message);
};
```



```
echo $example();

// Измененное значение в родительской области видимости
// отражается и внутри функции
$message = 'world';
echo $example();

// Замыканию также доступны и обычные аргументы
$example = function ($arg) use ($message) {
    var_dump($arg . ' ' . $message);
};
$example("hello");
?>
```



# Проверка существования функции

Функция с заданным или динамическим именем может быть доступна для вызова или нет. В связи с этим одной из важных задач является проверка существования функции. Эту задачу выполняют функции, представленные в таблице.

Функция	Описание
<code>function_exists(\$function_name)</code>	Возвращает TRUE, если функция с именем <code>\$function_name</code> существует, в противном случае возвращается FALSE
<code>get_defined_functions()</code>	Возвращает массив всех доступных в настоящий момент функций
<code>get_extension_funcs(\$module_name)</code>	Возвращает массив с именами функций для расширения <code>\$module_name</code>
<code>extension_loaded(\$name)</code>	Проверяет, подключено ли расширение с именем <code>\$name</code>
<code>get_loaded_extensions</code> ( <code>[\$zend_extensions = FALSE]</code> )	Возвращает массив подключенных расширений

Проверить существование одиночной функции проще всего при помощи функции `function_exists()`

## Пример

```
<?php
if (TRUE)
{
    function get_sum_if($fst, $snd)
    {
        return $fst + $snd;
    }
}

if (function_exists("get_sum")) // TRUE
    echo "Функция get_sum() существует<br>";
else
    echo "Функция get_sum() не существует<br>";

if (function_exists("get_sum_if")) // TRUE
    echo "Функция get_sum_if () существует<br>";
else
    echo "Функция get_sum_if () не существует<br>";
if (function_exists("get_sum_if_post")) // FALSE
    echo "Функция get_sum_if_post() существует<br>";
else
    echo "Функция get_sum_if_post() не существует<br>";
```



```

function get_sum($fst, $snd)
{
    return $fst + $snd;
}
if (TRUE)
{
    function get_sum_if_post($fst, $snd)
    {
        return $fst + $snd;
    }
}
?>

```

Результатом выполнения скрипта будут следующие строки:

```

Функция get_sum() существует
Функция get_sum_if () существует
Функция get_sum_if_post() не существует

```

При помощи функции *get\_defined\_functions()* можно получить массив всех функций, доступных на текущий момент

```

<?php
function get_sum($fst, $snd)
{
    return $fst + $snd;
}

$arr = get_defined_functions();
echo "<pre>";
print_r($arr);
echo "</pre>";
?>

```

Функция *get\_defined\_functions()* возвращает двумерный ассоциативный массив, первый элемент *internal* содержит предопределенные функции из ядра PHP и расширений, второй элемент *user* содержит функции, определенные пользователем:

```

Array
(
    [internal] => Array
        (
            [0] => zend_version
            [1] => func_num_args
            [2] => func_get_arg
            [3] => func_get_args
            [4] => strlen
            [5] => strcmp
            [6] => strncmp
            ...

```

```
[1309] => socket_getopt
[1310] => socket_setopt
)

[user] => Array
(
    [0] => get_sum
)
)
```

Функция `get_extension_funcs()` позволяет вернуть массив с именами функций, принадлежащих расширению, имя которого передается в качестве параметра. Приведенный ниже скрипт возвращает список функция, которые определены в расширении GD.

```
<?php
// Получаем массив с именами функций для работы с MySQL
$arr = get_extension_funcs("gd");
echo "<pre>";
print_r($arr);
echo "</pre>";
?>
```

Результатом выполнения скрипта будет следующий дамп массива `$arr`:

```
Array
(
    [0] => gd_info
    [1] => imagearc
    [2] => imageellipse
    [3] => imagechar
    ...
    [98] => imagecolormatch
    [99] => imagefilter
    [100] => imageconvolution
)
```

Для того чтобы функция `get_extension_funcs()` вернула результат, соответствующее расширение должно быть подключено в конфигурационном файле `php.ini`. Если расширение не подключено, функция вернет `FALSE`. Для проверки факта, подключено расширение или нет, предназначена специальная функция `extension_loaded()`, которая принимает в качестве параметра имя расширения и возвращает `TRUE`, если оно подключено, и `FALSE` в противном случае.

## Вспомогательные функции

Ниже приведен список часто используемых вспомогательных функций, которые содержатся в ядре PHP.

Функция	Описание
<code>exit([\$status])</code>	Прекращает выполнение скрипта. Если указан необязательный параметр <code>\$status</code> , он выводится в окно браузера
<code>die()</code>	Синоним для функции <code>exit()</code>
<code>sleep(\$seconds)</code>	Осуществляет задержку выполнения скрипта на <code>\$seconds</code> секунд
<code>time_nanosleep(\$seconds, \$nanoseconds)</code>	Осуществляет задержку выполнения скрипта на <code>\$seconds</code> секунд и <code>\$nanoseconds</code> наносекунд
<code>usleep(\$micro_seconds)</code>	Осуществляет задержку выполнения скрипта на <code>\$micro_seconds</code> микросекунд
<code>eval(\$code)</code>	Выполняет PHP-код из строки <code>\$code</code>
<code>empty(\$var)</code>	Определяет, не пуста ли переменная <code>\$var</code> , возвращая <code>TRUE</code> , если переменная пустая, и <code>FALSE</code> в противном
<code>isset(\$var)</code>	Определяет, существует ли переменная <code>\$var</code> , возвращая <code>TRUE</code> , если переменная существует, и <code>FALSE</code> в противном случае
<code>unset(\$var [, \$var1 [, ...]])</code>	Уничтожает одну или более переменных, массивов, объектов, переданных конструкции в качестве параметров
<code>print_r(\$var [, \$return])</code>	Выводит дамп переменной, массива или объекта <code>\$var</code> . Если необязательный параметр <code>\$return</code> принимает значение <code>TRUE</code> , вместо вывода дампа в окно браузера, он возвращается в виде строки
<code>var_dump(\$var [, \$var1 [, ...]])</code>	Выводит дамп одной и большего количества переменных, включая их типы данных
<code>phpinfo()</code>	Выводит таблицы с параметрами текущей версии PHP, расширений и переменных окружений
<code>phpversion([\$extension])</code>	Возвращает текущую версию PHP или версию расширения, если необязательный параметр <code>\$extension</code> содержит его имя



<code>php_uname([\$mode])</code>	Возвращает сведения об операционной системе, в которой выполняется PHP. Необязательный параметр <code>\$mode</code> может принимать значения параметров для одноименной утилиты UNIX-подобных операционных систем
<code>getenv(\$varname)</code>	Возвращает значение переменной окружения с именем <code>\$varname</code> .