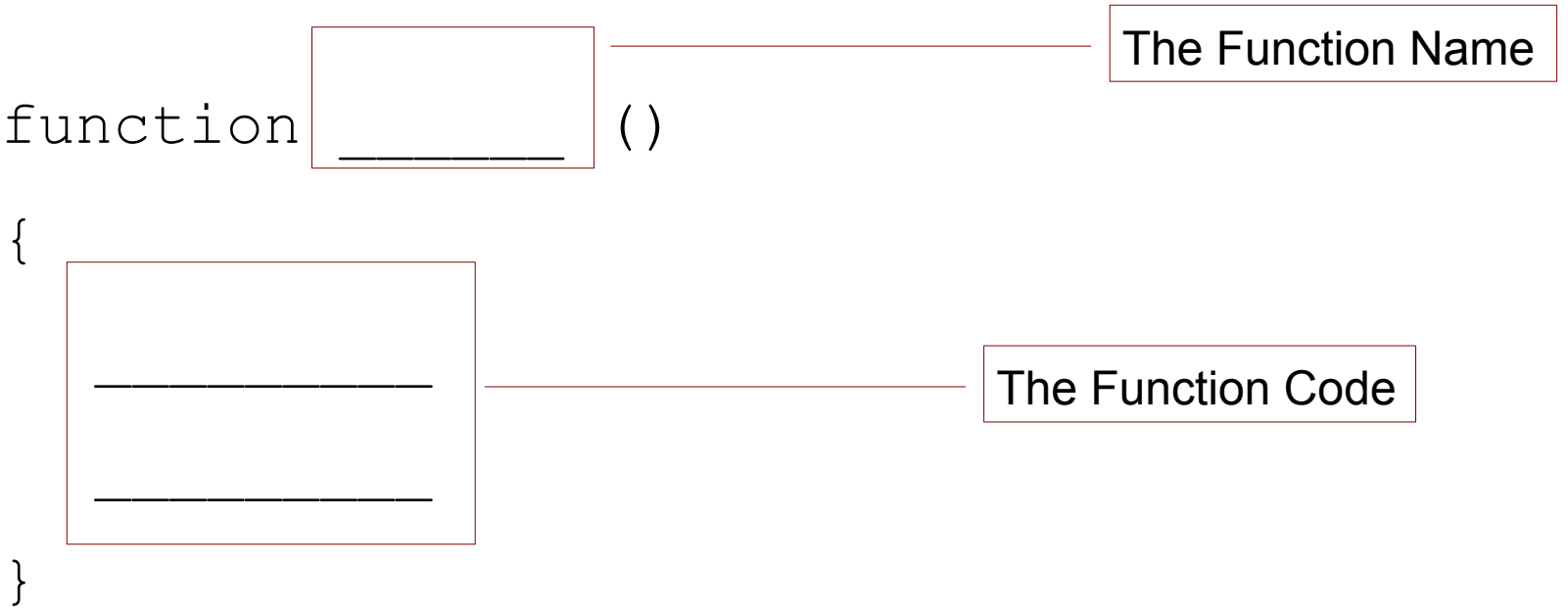
Functions

What is a Function?

- A function encapsulates piece of code in a way that allows it to be called again and again.

Basic Syntax

- The basic syntax we should follow when writing a function is:



Basic Syntax

- Example

```
function doSomething()
{
   echo "Something";
}
```

Basic Syntax

- The function name can consist letters, numbers and underscore only. The function name cannot start with a number.
- Calling a function is done by writing its name.

```
doSomething();  //prints out "Something"
```

Basic Syntax

- PHP functions are not case sensitive. We can mistype the function name writing a capital instead of a small letter and vice verso and the function will still be executed.

The Returned Value

- All PHP functions return a value. Unlike other languages, we don't include the type of the returned value in the function title.
- When using the 'return' keyword, the PHP engine identifies the type of the returned value

The Returned Value

```
function sum($numA,$numB)
{
    $sum = $numA + $numB;
    return $sum;
}
```

Function Parameters

- We can add parameter\s within the function title brackets. Unlike other languages, we just need to write a variable within those brackets. No need to specify the type.
- It is possible to pass any number of arguments regardless the number of parameters that were specified in the function declaration (as long as you don't pass fewer than the number of parameters that were specified).

Function Parameters

- It is possible to define parameters with a default value (for case an argument wasn't sent) by adding the equal sign and the default value.

```
<?php
function greet($name,$greeting="Good Morning")
{
    echo "$greeting $name";
}
greet("Moshe");
?>
```

Variable Number of Arguments

- PHP provides three built-in functions that assist handling variable length arguments list:

  `func_num_args()`

  This function returns the number of arguments.

  `func_get_arg()`

  This function returns the argument its indexed number was passed to (e.g. `func_get_arg(0)` returns the first argument, `func_get_arg(1)` returns the second argument etc...).

  `func_get_args()`

  This function returns an array that holds all arguments.

Variable Number of Arguments

```
<?php

function doSomething()
{
    $number_of_arguments = func_num_args();
    echo "<P>".$number_of_arguments;
    if($number_of_arguments!=0)
    {
          for($index=0; $index<$number_of_arguments; $index++)
          {
              echo "<br>".func_get_arg($index)
          }
    }
}
?>
```

Variable Number of Arguments

```
doSomething();
doSomething("Moshe");
doSomething("David","Moshe");
doSomething("Jane","Sam", "Rocky");

?>
```

Passing Arguments by Reference

- Prefixing a function parameter with the '&' operator will cause the argument to be sent “by reference” instead of “by value”.

Passing Arguments by Reference

```
<?php

    function doSomething(&$variable)
    {
        $variable++;
    }

    $number = 10;
    doSomething($number);
    echo $number;
?>
```

Returned Value (By Reference)

- PHP functions return values 'by value' (by default). Adding '&' to the function name shall result in returning a value 'by reference'.
- Suitable for cases in which we need to return resources (e.g. connection with DB).

Returned Value (By Reference)

```
function &getConnection()
{
    ...
    ...
    return $connection;
}
```

Variables Scope

- Global Scope

  Variables with the global scope are available to all parts of the script. Defining (or assigning a value to) a variable outside a function or class will set that variable to be with the “global” scope.

- Function Scope

  Each function has its own scope, completely isolated from the global scope. Any variable defined within the scope of a given function won't be available once the function finishes its execution.

Accessing Global Variables

- There are two possible ways to access a global variable from within a function:

  Importing

  We can import the global variable by using the global statement.

```
<?php
$name = "Moshe";
function hello()
{
    global $name;
    echo "Hello $name!";
}
hello();
?>
```

Accessing Global Variables

Using the `$GLOBALS` Super Global Array

The `$GLOBALS` array is accessible from everywhere. Using `$GLOBALS` it is possible to access any of the global variables. Writing the global variable name (without the '$' sign) within the brackets of `$GLOBALS` returns the value of that global variable.

```
<?php
$name = "Moshe";
function hello()
{
    echo "Hello $GLOBALS[name]!";
}
hello();
?>
```

Anonymous Functions

- As of PHP 5.4 it is possible to refer `$this` from within the scope of anonymous function.

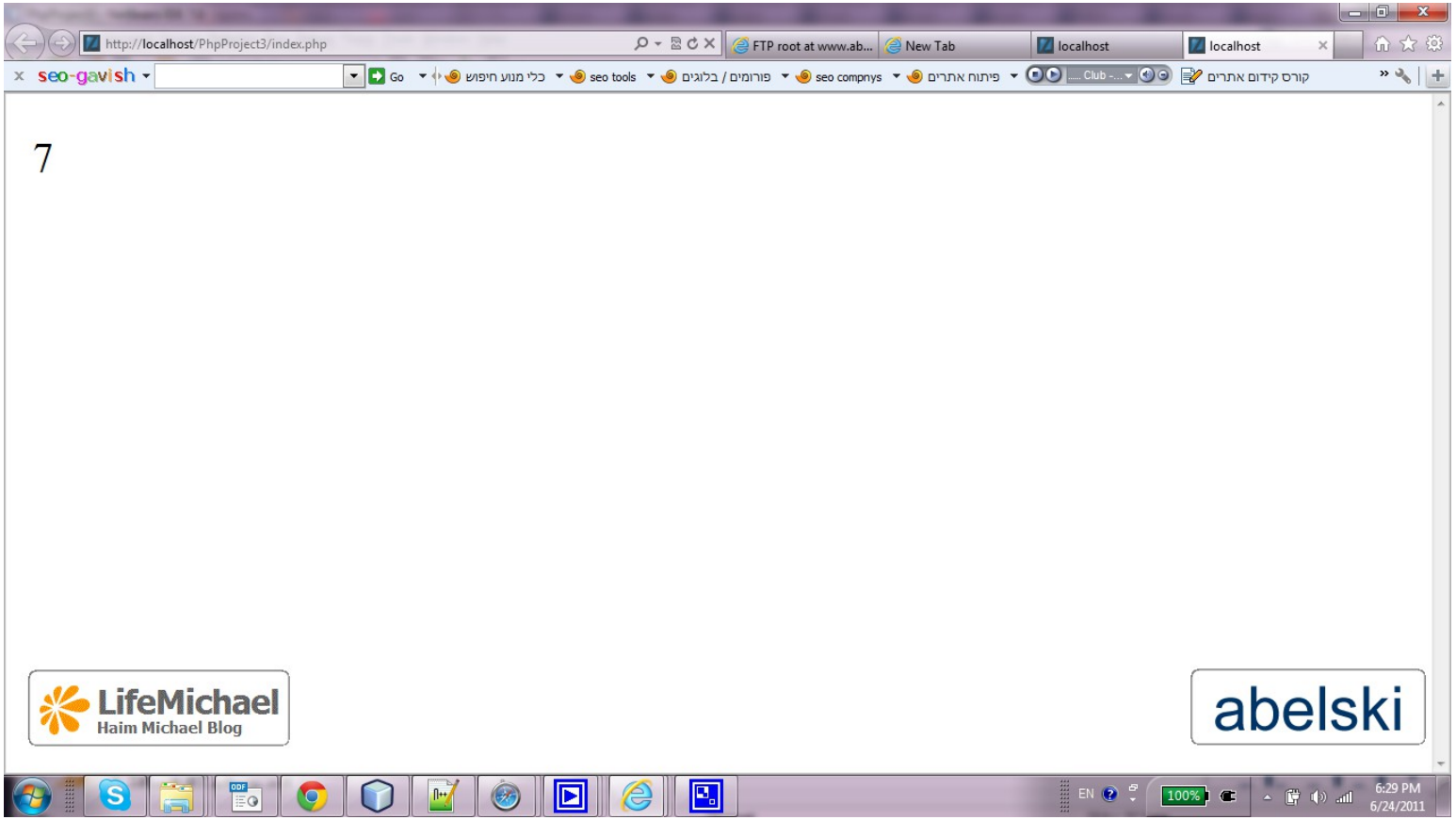
Anonymous Functions

- PHP allows us to define anonymous functions. We can assign the anonymous function to a variable and invoke it using that variable.

Anonymous Functions

```
<?php
$myfunc = function ($numA,$numB)
{
    return $numA+$numB;
};
echo $myfunc(3,4);
?>
```

Anonymous Functions



Anonymous Functions

- PHP 5.4 allows us to refer `$this` from within the scope of the anonymous function.

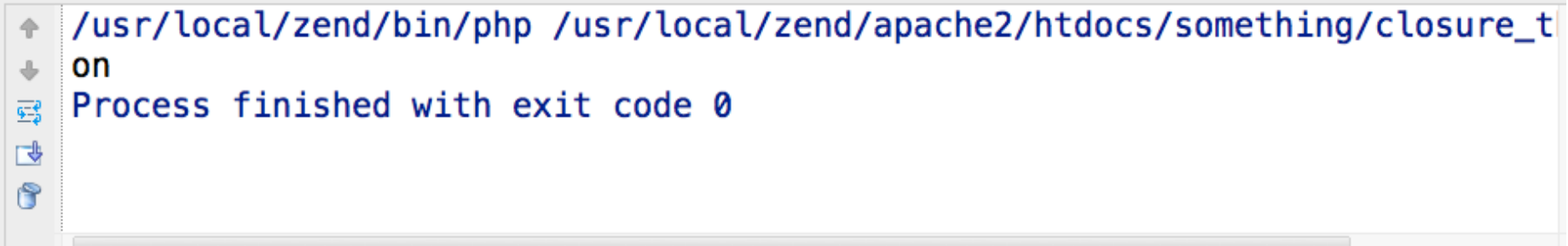
Anonymous Functions

```
<?php
class Car
{
    var $mode;
    var $engine;
    function __construct()
    {
        $this->engine = new Engine();
    }
    function start()
    {
        $start_engine = function()
        {
            $this->engine->mode="on";
        };
        $start_engine();
    }
}
```

Anonymous Functions

```
class Engine
{
    var $mode;
}

$ob = new Car();
$ob->start();
echo $ob->engine->mode;
```

Anonymous Functions



Variadic Functions

- PHP 5.6 provides us with the … operator, that allows us to create variadic functions.
- Variadic functions are functions with a varied number of parameters in according with the number of arguments passed over.

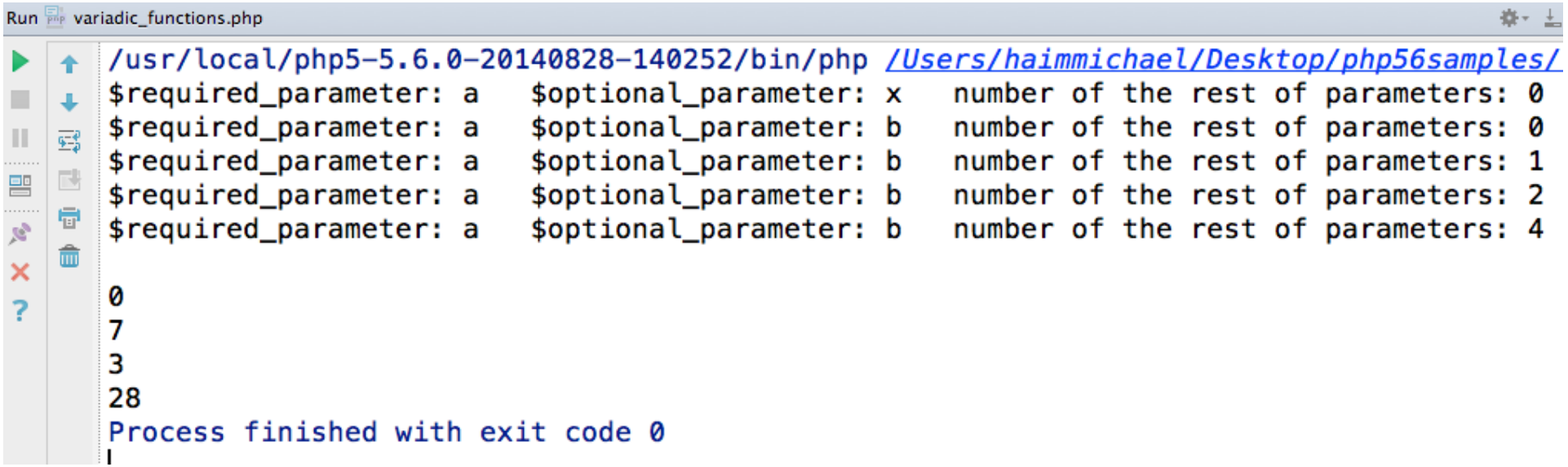
Variadic Functions

```
<?php
function func($required_parameter, $optional_parameter = 'x', ...
$the_rest_of_the_parameters)
{
    // $params is an array that contains the rest of the arguments
    printf('$required_parameter: %s   $optional_parameter: %s   number
        of the rest of parameters: %d'."\n",
        $required_parameter,
        $optional_parameter,
        count($the_rest_of_the_parameters));
}

function sum(...$numbers)
{
    $total = 0;
    foreach($numbers as $number)
    {
        $total += $number;
    }
    return $total;
}
```

Variadic Functions

```
func('a');
func('a', 'b');
func('a', 'b', 'c');
func('a', 'b', 'c', 'd');
func('a', 'b', 'c', 'd', 'e', 'zzz');

echo "\n".sum();
echo "\n".sum(7);
echo "\n".sum(1,2);
echo "\n".sum(1,2,3,4,5,6,7);
?>
```

Variadic Functions



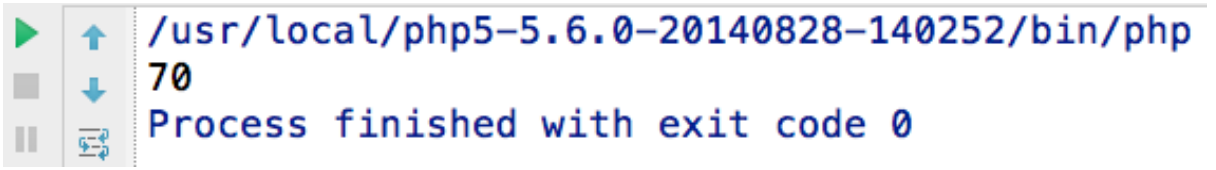
Arguments Unpacking

- As of PHP 5.6 we can use the '…' operator for passing over an array of values to a function we call and have the array's values unpacked and assigned to the function parameters.

Arguments Unpacking

```
<?php
function total($a,$b,$c)
{
    return $a+$b+$c;
}

$vec = [12,4,54];

echo total(...$vec);
?>
```

Arguments Unpacking



Scalar Type Declaration

- As of PHP 7, when declaring a function we can now specify type for each one of the parameters. We can specify any of the following scalar types: `string`, `int`, `float` or `bool`.
- These types come in addition to the types we could already use as of PHP 5.x including a class name, an interface name, `array` or `callable`.

Scalar Type Declaration

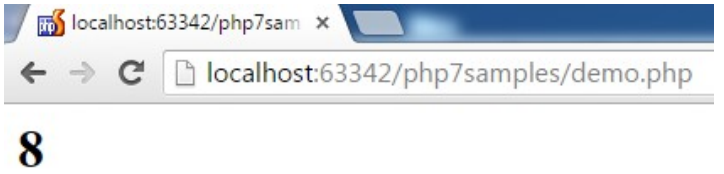
- When specifying the types of a function parameters we can either do it in a coercive mode (default) or a strict mode.
- In order to be in a strict mode we should add the `declare()` directive to the beginning of the file. When in strict mode, if the type check fails then a TypeError exception will be thrown.

Scalar Type Declaration

- The one exception for this behavior is when assigning a value of the type `int` to parameter of the type `float`.

Scalar Type Declaration

```php
<?php

declare(strict_types=1);

function sum(float $a, float $b)
{
    return $a+$b;
}

$temp = sum(3,5);

echo "<h1>$temp</h1>";
?>
```



Return Type Declaration

- As of PHP 7, we can specify the type of the returned value for the function we declare.
- The return type can be `string`, `int`, `float`, `bool`, `array`, `callable`, `self` (when defining methods only), `parent` (when defining methods only), `Closure`, the name of a class or the name of an interface.

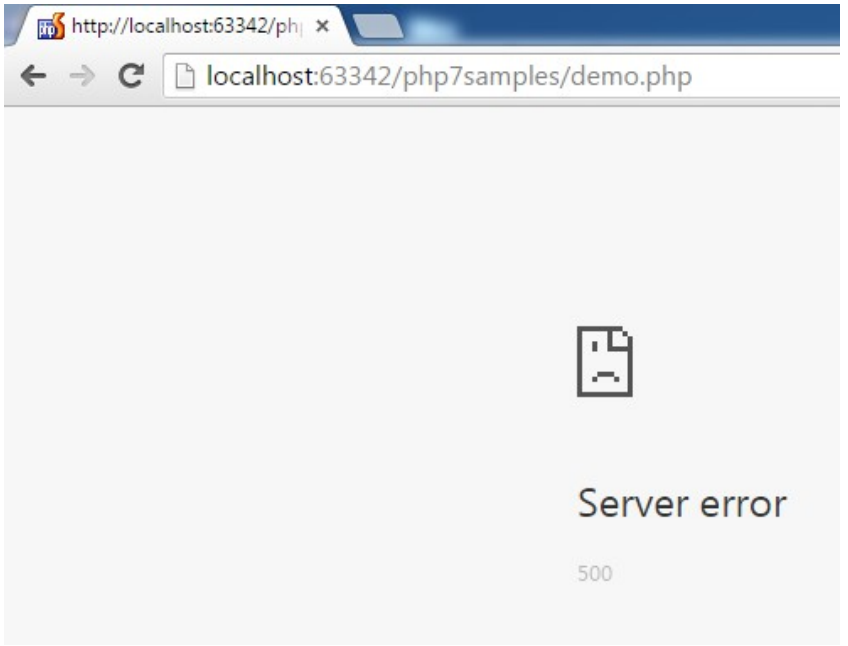
Return Type Declaration

- When overriding a method, the type of the returned value of the new defined method must be the same as of the overridden method.

Return Type Declaration

```
<?php

declare(strict_types=1);

function sum(int $a, int $b):float
{
    return $a+$b;
}

$temp = sum(sum(5,2),5);

echo "<h1>$temp</h1>";
?>
```

Return Type Declaration



The CSPRNG Functions

- CSPRNG stands for Cryptographically Secure Pseudo Random Number Generator. As of PHP7, we can use two new functions for generating cryptographically secure integers and strings.

```
string random_bytes(int length);

int random_int(int min, int max);
```

- These two functions emits an Error exception if the source of sufficient randomness cannot be found.

The CSPRNG Functions

```
<?php
$str = random_bytes(10);
$num = random_int(0, 10);
echo "<h3>random string</h3>";
foreach (str_split($str) as $chr) {
    echo "<br/>".ord($chr);
}
echo "<br/> <br/> <br/>";
echo "<h3>random number</h3>";
echo "<br/>num=$num";
?>
```

The CSPRNG Functions