# Programming with Databases

## PHP

PHP has a reputation as a web-scripting language.

However, it also works as a general-purpose scripting language.

Later versions (since PHP5) also have a strong object model.

Undeserved reputation: toy, poorly-designed language.

Poor design may be true of some PHP libraries.
The language itself, however, has many good aspects.

## ... PHP

PHP scripts consist of

```
#!/usr/bin/php
<?
... PHP code ...
?>
```

(#! line is optional)

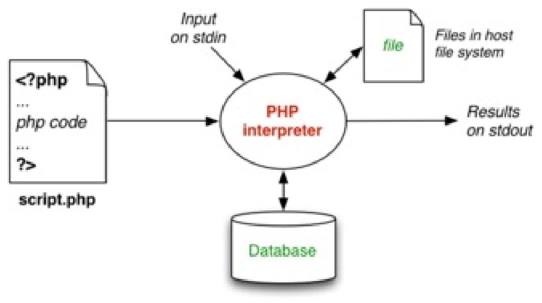
Can be executed from command-line via:

```
$ php script.php
$ chmod 755 script.php
$ ./script.php
```

`$argv[]` contains command-line parameters.

## ... PHP

Execution environment of PHP scripts:



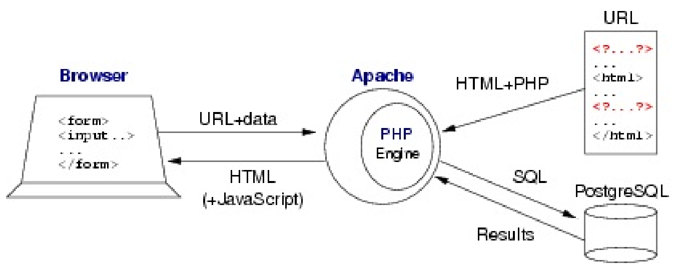
## ... PHP

PHP web scripts are a mixture of HTML and PHP code ...

- stored on web server (Apache) under its DocumentRoot
- invoked via URL (`http://server/a/b/script.php`)
- parameters passed either via GET or PUT
- executed in an engine (Zend) inside the web server
- with environment/privileges of web server process
- having access to cookies (on client) and DBMSs (on server)

## ... PHP

Architecture of typical Apache/PHP server:



## From PHP to HTML

How the PHP engine treats a script:

- scan the script from top to bottom; interpolate `require`d files
- any text *not* enclosed in `<?...?>` is copied to output
- any PHP expression enclosed in `<?=`*Expr*`?>` is evaluated, and its string representation is copied to output
- any PHP code enclosed in `<?`*Statements;*`?>` is executed, and any output it produces is sent to output
- first output is is preceded by HTTP header: `Content-type: text/html`
- `header()` function can be used to produce alternative HTTP header (if any output has already been sent when `header()` called, produces error)

## ... From PHP to HTML

Example PHP/HTML script:

```
<html>
<? require("myDefinitions.php");
    $pageName = $_POST["name"]; $max = $_POST["max"];
?>
<body bgcolor='purple'>
<h1>This is <?=$pageName?></h1>
<? if ($max <= 0) { ?>
   <b>There are no numbers to display</b>
<? }
else {
    for ($i = 0; $i <= $max; $i++)
        echo "$i<br>\n";
}
?>
</body>
</html>
```

## ... From PHP to HTML

Nowadays, most PHP usage in web application frameworks*

- using MVC design pattern
- providing overarching control of web app (C=control)
- template-based HTML rendering (V=view)
- providing DBMS-independent DB access (M=model)
  - often via Active Record pattern
  - and also providing SQL constructing functions
- and, of course, interface to JavaScript & CSS libraries

* e.g. CodeIgniter, Yii, Symfony, Laravel, Zend, CakePHP, .....

## The PHP Language

The PHP language has the following characteristics:

- C-like syntax (with some Perl flavour)
- "loose" attitude to types (determined by context)
- very easy to manipulate strings
- associative arrays (cf. Perl's hashes)
- extensive libraries of functions (2000 page manual)
- supports object-orientation (cf. Perl)
- comments introduced via # or //

Syntactically: "a simpler, more uniform version of Perl".

## ... The PHP Language

When PHP programs are executed in a Web server ...

The HTTP request supplies the parameters.
(or they're available in `$argv[]` if run from the command-line)

CGI params available in arrays $_GET, $_POST, $_REQUEST.

Example:

`http://server/user/list.php?name=John&age=21`

In the script, the parameters would be accessed as:

```
print "Name is $_REQUEST[name]\n";
print "Age is ".$_REQUEST['age']."\n";
print "Name: $_GET[name] Age: $_GET[age]\n";
```

## Variables

No variable declarations are required.

Variables are created by assigning a value to them.

All variable names are preceded by `$` (note: `$i`, `$i++`, `$++i`)

The type of a variable is that of the last assigned value.

Check/set variable *type* via `gettype`/`settype` functions.

Convert variable *value* via casting (e.g. `(int)`, `(string)`, ...)

Default value of unassigned variable is `null` (distinguished constant)
(if unset variable used, get `0` or `""` or `false`, depending on context, and error in log)

## ... Variables

**Examples:**

```
$foo = 3;          # $foo is an int, value 3

$foo = "8";        # $foo is now a string, value "8"

$foo = $foo + 2;   # $foo is now an int, value 10

$foo = "$foo green bottles";
                   # $foo is now "10 green bottles"

$foo = 3.0 * $foo; # $foo is now double, value 30.0

$foo = (int)$foo;  # $foo is now an int, value 30
```

## ... Variables

Lifetime of all variables is the current script.

Variables defined outside any function:

- have global scope (over whole to script)
- but are not accessible within functions unless "requested":

  ```
  function f() { global $max_num, $colour; .... }
  ```

"Super-global" arrays (e.g. `$_GET`, `$_PUT`, `$_SERVER`, `$_COOKIE`, ...):

- contain "environment" values (CGI params, server ENV, request data)
- are accessible from anywhere in the script

## Constants

Constants are defined using the `define()` function

- may only evaluate to scalar type values (e.g. int,float,string)
- have case-senstive names; written without dollar sign ($)
- are always available globally (like super-globals)
- may not be redefined or undefined once they have been set

```
define("CONSTANT", "Hello world.");
define("MaxLevel", 6);
echo CONSTANT; // outputs "Hello world."
echo Constant; // outputs "Constant" and gives error
if ($i > MaxLevel) { echo "Yes"; }
```

## Types

Four scalar types:

- boolean, with values `true` and `false` (case-insensitive)
  - uses C-like interpretation for `false` (i.e. `0`, `""`, ...)
  - all non-zero values are treated as `true`
    (beware: this includes negative error status values)
- integer, e.g. `0`, `1`, `-999`, ... (standard 32-bit int format)
- float, e.g. `3.14`, `2.0e6`, ... (IEEE floating point format)

* string ... (see next slide)

---

## Strings

Strings: sequences of characters, similar to Perl

* double-quoted strings ("...") permit interpolation, e.g.

```
$x = 5;  print "Value of x is $x\n";
// prints "Value of x is 5"
```

* single-quoted strings ('...') don't do interpolation

```
$x = 5;  print 'Value of x is $x\n';
// prints "Value of x is $x"
```

* non-quoted "strings" (abc)   (only work in some contexts)

Notes:
* non-quoted strings look like C/Java variables; PHP variables look like $abc
* non-quoted strings are actually an error; normally used for constants;
  in some contexts they produce a value which is the same as their name

---

## ... Strings

Strings (cont)

"heredoc" strings available for large multi-line strings

```
print <<<XYZ
This is a "here" document. It can contain
many lines of text, with interpolation.
Such as the value of x is $x
With any old "quotes' the we ``like''
XYZ;

$str = <<<aLongString
This is my "long" string.
Ok, it's not really so long
aLongString;
```

---

## ... Strings

When variables are used inside a "..." string or heredoc

* their value is interpolated into the string
* after being converted to a suitable string representation

Example:

```
$a = 1;  $b = 3.5;  $c = "Hello";
$str = "a:$a,  b:$b,  c:$c";
// now $str == "a:1, b:3.5, c:Hello"
```

---

## ... Strings

Rules for interpolation and escape sequences:

| | |
|---|---|
| `"..."` | must escape embedded " via \"<br>escape sequences work<br>variable interpolation works |
| `'...'` | no variable interpolation<br>no escape sequences work (including no \') |
| heredoc's | no need to escape embedded "<br>escape sequences work<br>variable interpolation works |

PHP escape sequences are like C/Java/Perl e.g. \n, \t, ...

---

## ... Strings

Note that interpolation does occur in  `"This is '$it'"`

I.e. `<? $it = 5; print "This is '$it'"; ?>` displays `This is '5'`

This is important in producing HTML in PHP since attribute values for HTML tags should be quoted.

**Example**: we want to create a text input box to collect a new value for parameter name, and display its current value:

```
print "<input type='text' name='qty' value='$_GET[qty]'>\n";
```

Note: If the `qty` parameter is not set, then the `$_GET["qty"]` will have no value, and the text box will be empty.

## ... Strings

Other operations on strings:

`.` (dot) for string concatentation (cf. Perl's `.` or Java's `+`)

```
$x = 127;
print "Result is ".sqrt($x)."\n";
```

`trim()` removes whitespace from left and right ends of string

```
// $s == "  blah  blah     "
$s = trim($s);
// $s == "blah  blah"
```

## ... Strings

More operations on strings:

`preg_split()` partitions string into array via Perl regexp

```
// $s == "  ab  cde fg"
$a = preg_split('/\s+/',$s);
// $a[0]=="" && $a[1]=="ab"
// && $a[2]=="cde" $a[3]=="fg"
```

`join()` assembles strings from an array

```
// $a[0]=="" && $a[1]=="ab"
// && $a[2]=="cde" $a[3]=="fg"
$s = join(":",$a);
// $s == ":ab:cde:fg";
```

Plus many others ... see PHP Manual for details.

## Arrays

PHP arrays = sequence of values accessible via index.

Indexes can be values of any scalar type, incl. strings.

This provides both scalar and associative arrays (hash tables).

E.g.

```
$a[0] = "abc";  $a[1] = 'def';  $a[2] = ghi;

$b['abc'] = 0;  $b[def] = 1;    $b["ghi"] = 2;
```

PHP arrays are like *ordered* hash-tables.

## ... Arrays

Arrays can be initialised element-at-a-time:

```
$word[0]="a";  $word[1]="the";  $word[2]="this";

$mark["ann"]=100;  $mark["bob"]=50;  $mark["col"]=9;

$vec[]=1;  $vec[]=3;  $vec[]=5;  $vec[]=7;  $vec[] = 9;
// which is equivalent to
$vec[0]=1;  $vec[1]=3;  $vec[2]=5;  $vec[3]=7; $vec[4] = 9;
```

Arrays can be initialised in a single statement:

```
$word = array("a", "the", "this");

$marks = array("ann"=>100, "bob"=>50, "col"=>9);

$vec = array(0 => 1, 1 => 3, 2 => 5, 3 => 7, 4 => 9);
// which is equivalent to
```

```
$vec = array(1, 3, 5, 7, 9);
```

## ... Arrays

Multiple values can be extracted from arrays via `list()`:

```
$a = array(5, 4, 3, 2, 1);
list($x,$y,$z) = $a;
# $x==5, $y==4, $z==3
```

Multi-dimensional arrays work ok (array elements can be any type)

```
$fruits = array ( "fruits"  => array ( "a" => "orange"
                                     , "b" => "banana"
                                     , "c" => "apple"
                                     )
                , "numbers" => array ( 1,2,3,4,5,6 )
                , "holes"   => array (      "first"
                                     , 5 => "second"
                                     ,      "third"
                                     )
                );
```

## ... Arrays

Several mechanisms are available for iteration over arrays:

```
for ($i = 0; $i < count($word); $i++)
   print "word[$i] = $word[$i]\n";

foreach ($words as $w) print "next word = $w\n";

for (reset($marks); $name = key($marks); next($marks))
   print "Mark for $name = $marks[$name]\n";

reset($marks);
while (list($name,$val) = each($marks))
   print "Mark for $name = $val\n";

$elem = current($vec);
while ($elem) {
   print "Next elem is $elem\n";
   $elem = next($vec);
}
```

First method only works if indexes are integers; missing values returned as `null`.

## ... Arrays

Example: iterating over an array:

```
$marks = array("Ann"=>95, "John"=>75, "David"=>60);

foreach ($marks as $name => $mark)
        echo "$name scored $mark%\n";

echo "Whole array: $marks\n";
```

which displays:

```
Ann scored 95%
John scored 75%
David scored 60%
Whole array: Array
```

## Other PHP Types

PHP has standard notion of *class*: data values plus methods

```
// creating an object of class foo
$x = new foo;  $x->method(1,'a');
```

Resource: special type for references to external resources

- e.g. database connections/cursors, file handles, ...

NULL: a distinguished value NULL (or `null`, case-insensitive)

- used to indicate that a variable exists but has no value

## Variable Checking

Functions to test properties of a variable ...

`isset($v)` ... $v has a non-NULL value
(can check whether an array has a value for a given index)

`is_null($v)` ... $v has the value NULL

`empty($v)` ... $v has value NULL or 0 or "" or array()

`unset($v)` ... effectively removes variable $v

---

## Variable Variables

PHP provides a way to dynamically create variable names.

Example:

```
for ($i = 0; $i < $MAX; $i++) {
   $varname = "myVar$i";
   $value   = ${$varname};
   print "Value of $varname = $value\n";
}
```

Accesses variables called `myVar0`, `myVar1`, `myVar2`, ...

Note: this is *not* the same as an array `myVar[0]`, `myVar[1]`, `myVar[2]`, ...

Useful in e.g. HTML forms, where we may have a collection of variables that can't be represented by an array, but we need to iterate over them ...

---

## Control Structures

Control structures have similar syntax to C/Perl/Java.

```
{ Statement1; Statement2; ... }

if (Expression1) Statement1
[elseif (Expression2) Statement2 ...]
[else Statementn]

switch (Expression1) {
case Value1: Statement1; break; ...
[case Value2: Statement2; break; ...]
}

while (Expression) Statement
for (Init; Test; Next) Statement
foreach (ArrayVar as [KeyVar =] ValVar) Statement
```

---

## Functions

Functions are defined as:

```
function FuncName($arg1, $arg2, ... )
{
   Statement; ...
   return Expression;
}
```

Example:

```
// return array of first n integers
function iota($n)
{
   for ($i = 1; $i <= $n; $i++)
      $list[] = $i;
   return $list;
}
```

---

## ... Functions

Notes on function definitions:

- don't specify argument types or return type
- can specify default values for arguments
  - can omit arguments from right-to-left if default values given
  - if no defaults are given, missing arguments generate errors
- can handle variable-length argument lists (like C's printf)
  - using special functions `func_num_args()`, `func_get_arg()`, and `func_get_args()`

## ... Functions

Example for default parameter values:

```
function makeCoffee($type="latte", $size="big") {
    return "Making a $size cup of $type.\n";
}
echo makeCoffee();
echo makeCoffee("cappucino");
echo makeCoffee("espresso","tiny");
```

which will display

```
Making a big cup of latte.
Making a big cup of cappucino.
Making a tiny cup of espresso.
```

## ... Functions

Example for variable length argument lists:

```
function foo() {
    $numargs = func_num_args();
    echo "Number of args: $numargs\n";
    if ($numargs >= 2)
        echo "Second arg is: ",func_get_arg(1),"\n";
    $args = func_get_args();
    for ($i = 0; $i < $numargs; $i++)
        echo "Arg$i = $args[$i]  ";
}
foo(1, 'b', 3);
```

which will display

```
Number of args: 3
Second arg is: b
Arg0 = 1  Arg1 = b  Arg2 = 3
```

## Debugging

`print_r($v)` displays representation of $v's value

`var_dump($v)` displays more info on $v's value

`error_reporting(`*Level*`)` controls how much error display

`@func()` executes `func()` and supresses error reporting

## PHP and Databases

To interact with databases, PHP needs ways to:

- establish a connection with a database (authentication)
- construct SQL statements from program values
- send SQL statements to the DBMS for execution
- for updates, check how many tuples affected
- for queries, iterate through the result tuples
- extract fields from returned tuples

Different database libraries all handle these slightly differently.

## PHP and PostgreSQL

PHP has a library of functions for PostgreSQL interaction.

Follow typical PL/DBMS interaction pattern:

- send SQL query, retrieve results one-at-a-time
- access to database and result-set metadata

Obvious problem: code written using it is non-portable.

There is also a generic DB-access library called PDO.

## ... PHP and PostgreSQL

Most DB applications can be handled with just five functions:

- `pg_connect()` ... connect to the database
- `pg_query()` ... send SQL statement for processing
- `pg_fetch_array()` ... retrieve the next result tuple
- `pg_num_rows()` ... count # rows in result
- `pg_affected_rows()` ... count # rows changed

---

## The `pg_connect` Function

`resource` **`pg_connect`**`(string` *ConnParams*`)`

- attempts to connect to database specified in *ConnParams*
- precise format of *ConnParams* depends on configuration, e.g.

```
$db = pg_connect("dbname=mydb");
# or
$cp = "dbname=hisdb user=fred password=abc";
$db = pg_connect($cp);
```

- returns a `resource`, which is used for DB interactions
- if any problems, returns 0 (illegal connection)
  - possible problems: invalid password, unknown DB, ...
- the `pg_last_error()` function gives details of any error

---

## The pg_query Function

`resource` **`pg_query`**`(resource` *db*`, string` *Stmt*`)`

- sends the SQL statement *Stmt* to the database *db*
- *Stmt* can be either a query or insert/delete/update
- returns a `resource`, which is either
  - a cursor on the result set for query
  - nothing useful for insert/delete/update
- if any problems, returns 0 (illegal cursor)
  - possible problems: invalid *db*, syntax errors in *Stmt*
- subsequent attempts to use illegal cursor give PHP error

---

### ... The pg_query Function

Example:

```
$unidb  = pg_connect("dbname=UniDB");
$query  = "select name
           from Staff where dept=2";
$result = pg_query($unidb, $query);
if (!$result)
   print "Something wrong with query!\n";
else
   // process the result set ...
```

To find out exactly what was wrong with the query ...

```
if (!$result)
   print pg_last_error();
```

---

## The `pg_num_rows` Function

`int` **`pg_num_rows`**`(resource` *Result*`)`

- returns the number of tuples in a `pg_query` query result
- zero, if the `pg_query` statement was an update

Example:

```
$query = "select * from Employees
          where department='Sales'";
$result = pg_query($db, $query);
if (!$result)
   print pg_last_error();
else if (pg_num_rows($result) > 20)
   print "This is a very big department\n";
```

---

## The `pg_affected_rows` Function

`int` **`pg_affected_rows`**`(resource` *Result*`)`

- returns # modified tuples in a `pg_query` update
- zero, if the `pg_query` statement was a query

Example:

```
$query = "delete from Enrolments
          where course='COMP3311'";
$result = pg_query($db, $query);
if (!$result)
   print pg_last_error();
else  {
   $nstudes = pg_affected_rows($result);
   print "Dropped $nstudes from COMP3311\n";
}
```

## The `pg_fetch_row` Function

`array` **`pg_fetch_row`**`(resource` *Res*`, int` *which*`)`

- fetches the $i^{th}$ tuple in a query result set
- if no *which* argument, fetches next tuple
- returns an `array` value that can be treated as a result row
- fields are accessed by position; based on query select list
- if no more elements left, returns 0

## ... The `pg_fetch_row` Function

Example:

```
$query = "select id,name from Staff";
if ($result = pg_query($db, $query)) {
   $n = pg_num_rows($result);
   for ($i = 0; $i < $n; $i++) {
      $item = pg_fetch_row($result,$i);
      print "Name=$item[1], StaffID=$item[0]\n";
   }
}
```

## The `pg_fetch_array` Function

`array` **`pg_fetch_array`**`(resource` *Res*`, int` *which*`)`

- fetches the *i* th element (tuple) in a query result set
- returns an `array` value that can be treated as a result row
- array is indexed by field *names* as well as position
- if no more elements left, returns 0
- if no *which* argument, fetches next tuple

## ... The `pg_fetch_array` Function

Example:

```
$query = "select id,name from Staff";
if (!($result = pg_query($db, $query)))
   print "Error: ".pg_last_error();
else {
   $n = pg_num_rows($result);
   for ($i = 0; $i < $n; $i++) {
      $item = pg_fetch_array($result,$i);
      $nm = $item["name"]; $id = $item["id"];
      print "Name=$nm, StaffID=$id\n";
   }
}
```

## COMP3311 Database Library

Problems:

- constructing SQL statements from user-supplied data
- providing DBMS-independent interface to database
- handling transactions over multiple DB operations

We define a small libary that solve the first two.

More sophisticated libraries (e.g. PDO) solve all three.

The third can often be solved via stored procedures.

## ... COMP3311 Database Library

Functions in the COMP3311 database library:

- accessDB(dbname): establish connection to DB
- dbQuery(db,sql): send SQL statement for execution
- dbNext(res): fetch next tuple from result set
- dbOneTuple(db,sql): run SQL to get a single tuple
- dbOneValue(db,sql): run SQL to get a single value
- dbUpdate(db,sql): send SQL insert/delete/update
- mkSQL(fmt,v1,v2,...): build an SQL statement string

All functions terminate if an error occurs (debugging).

---

## ... COMP3311 Database Library

Standard pattern for extracting data from DB:

```
$db = dbConnect("dbname=myDB");
...
$min = ...;
$query = "select a,b,c from R where c >= %d";
$result = dbQuery($db, mkSQL($query, $min));
while ($tuple = dbNext($r)) {
        list($a,$b,$c) = $tuple;
        $tmp = $a - $b - $c;
        # or ...
        $tmp = $tuple["a"] - $tuple["b"]
                           - $tuple["c"];
}
...
```

---

## ... COMP3311 Database Library

My conventions for writing PHP/DB code:

- $q is the SQL query template (also for updates)
- $r the query result handle (a PHP resource)
- $t is the current tuple (array indexed by position and name)
- invoking a query: `$r = dbQuery($db, mkSQL($q,`*vars*`));`
- extracting fields: `list($a,$b,$c,...) = $t;`
- will also sometimes use: `$a = $t["a"];` ...

You don't have to follow these, but this is what examples look like.

---

## ... COMP3311 Database Library

`string mkSQL(string` *QueryTemplate*`, any` *$v_1$*`, any` *$v_2$*`, ...)`

- queries are often constructed by interpolating variables
- ensures that values are appropriately quoted/escaped
- uses `printf`-like mechanism for specifying interpolated values

Example:

```
$name = "O'Brien";
$tmpl = "select * from Employees".
        "where name = %s and salary > %d";
$qry = mkSQL($tmpl, $name, 50000);
// which produces the query string
select * from Employees
where name = 'O''Brien' and salary > 50000
```

---

## ... COMP3311 Database Library

Example of use:

```
$db = accessDB("mymyunsw");
$q = <<_SQL_
select s.sid, p.name
from   Students s, People p, Courses c,
       Subjects su, CourseEnrolments e, Terms t
where  s.id = p.id and e.student = s.id and
        e.course = c.id and c.subject = su.id
        and su.code = %s and c.term = t.id
        and t.year = %d and t.sess = %s
order by s.sid
_SQL_;
```

```
$sql = mkSQL($q, $subj, $year, $sess);
$r = dbQuery($db, $sql);
while ($t = dbNext($r)) echo "$t[sid] $t[name]\n";
```

---

Produced: 20 Mar 2019