Generators

- Generators overview
  - Generator objects
- Generator syntax
  - yield keyword
    - Yielding values with keys
    - Yielding null values
    - Yielding by reference
    - Generator delegation via yield from
- Comparing generators with Iterator objects

Generators overview

Generators provide an easy way to implement simple iterators without the overhead or complexity of implementing a class that implements the `Iterator` interface.

A generator offers a convenient way to provide data to ENTITY loops without having to build an array in memory ahead of time, which may cause the program to exceed a memory limit or require a considerable amount of processing time to generate. Instead, a generator function can be used, which is the same as a normal function, except that instead of returning once, a generator can ENTITY as many times as it needs to in order to provide the values to be iterated over. Like with iterators, random data access is not possible.

A simple example of this is to reimplement the `range` function as a generator. The standard `range` function has to generate an array with every value in it and return it, which can result in large arrays: for example, calling `range(0, 1000000)` will result in well over 10 MB of memory being used.

As an alternative, we can implement an `xrange()` generator, which will only ever need enough memory to create an `Iterator` object and track the current state of the generator internally, which turns out to be less than 1 kilobyte.

Implementing `range` as a generator

```php
<?php
function xrange($start, $limit, $step = 1) {
    if ($start <= $limit) {
        if ($step <= 0) {
            throw new LogicException('Step must be positive');
        }

        for ($i = $start; $i <= $limit; $i += $step) {
            yield $i;
        }
    } else {
        if ($step >= 0) {
            throw new LogicException('Step must be negative');
        }

        for ($i = $start; $i >= $limit; $i += $step) {
            yield $i;
        }
    }
}

/*
 * Note that both range() and xrange() result in the same
 * output below.
 */

echo 'Single digit odd numbers from range():  ';
foreach (range(1, 9, 2) as $number) {
    echo "$number ";
}
echo "\n";

echo 'Single digit odd numbers from xrange(): ';
foreach (xrange(1, 9, 2) as $number) {
    echo "$number ";
}
```

ENTITY

```
Single digit odd numbers from range():  1 3 5 7 9
Single digit odd numbers from xrange(): 1 3 5 7 9
```

`Generator` objects

When a generator function is called, a new object of the internal `Generator` class is returned. This object implements the `Iterator` interface in much the same way as a forward-only iterator object would, and provides methods that can be called to manipulate the state of the generator, including sending values to and returning values from it.

Generator syntax

A generator function looks just like a normal function, except that instead of returning a value, a generator ENTITYs as many values as it needs to. Any function containing ENTITY is a generator function.

When a generator function is called, it returns an object that can be iterated over. When you iterate over that object (for instance, via a ENTITY loop), PHP will call the object's iteration methods each time it needs a value, then saves the state of the generator when the generator yields a value so that it can be resumed when the next value is required.

Once there are no more values to be yielded, then the generator can simply return, and the calling code continues just as if an array has run out of values.

> [!NOTE] A generator can return values, which can be retrieved using Generator::getReturn.

`yield` keyword

The heart of a generator function is the `yield` keyword. In its simplest form, a yield statement looks much like a return statement, except that instead of stopping execution of the function and returning, yield instead provides a value to the code looping over the generator and pauses execution of the generator function.

A simple example of yielding values

```
<?php
function gen_one_to_three() {
    for ($i = 1; $i <= 3; $i++) {
        // Note that $i is preserved between yields.
        yield $i;
    }
}

$generator = gen_one_to_three();
foreach ($generator as $value) {
    echo "$value\n";
}
```

ENTITY

```
1
2
3
```

> [!NOTE] Internally, sequential integer keys will be paired with the yielded values, just as with a non-associative array.

Yielding values with keys

PHP also supports associative arrays, and generators are no different. In addition to yielding simple values, as shown above, you can also yield a key at the same time.

The syntax for yielding a key/value pair is very similar to that used to define an associative array, as shown below.

Yielding a key/value pair

```
<?php
/*
 * The input is semi-colon separated fields, with the first
```

```
 * field being an ID to use as a key.
 */

$input = <<<'EOF'
1;PHP;Likes dollar signs
2;Python;Likes whitespace
3;Ruby;Likes blocks
EOF;

function input_parser($input) {
    foreach (explode("\n", $input) as $line) {
        $fields = explode(';', $line);
        $id = array_shift($fields);

        yield $id => $fields;
    }
}

foreach (input_parser($input) as $id => $fields) {
    echo "$id:\n";
    echo "    $fields[0]\n";
    echo "    $fields[1]\n";
}
```

ENTITY

```
1:
    PHP
    Likes dollar signs
2:
    Python
    Likes whitespace
3:
    Ruby
    Likes blocks
```

Yielding null values

Yield can be called without an argument to yield a ENTITY value with an automatic key.

Yielding ENTITYs

```
<?php
function gen_three_nulls() {
    foreach (range(1, 3) as $i) {
        yield;
    }
}

var_dump(iterator_to_array(gen_three_nulls()));
```

ENTITY

```
array(3) {
  [0]=>
  NULL
  [1]=>
  NULL
  [2]=>
  NULL
}
```

Yielding by reference

Generator functions are able to yield values by reference as well as by value. This is done in the same way as returning references from functions: by prepending an ampersand to the function name.

Yielding values by reference

```
<?php
function &gen_reference() {
    $value = 3;

    while ($value > 0) {
        yield $value;
```

```
    }
}

/*
 * Note that we can change $number within the loop, and
 * because the generator is yielding references, $value
 * within gen_reference() changes.
 */
foreach (gen_reference() as &$number) {
    echo (--$number).'... ';
}
```

ENTITY

```
2... 1... 0...
```

Generator delegation via `yield from`

Generator delegation allows you to yield values from another generator, `Traversable` object, or `array` by using the `yield from` keyword. The outer generator will then yield all values from the inner generator, object, or array until that is no longer valid, after which execution will continue in the outer generator.

If a generator is used with `yield from`, the `yield from` expression will also return any value returned by the inner generator.

> [!CAUTION] `yield from` does not reset the keys. It preserves the keys returned by the `Traversable` object, or `array`. Thus some values may share a common key with another `yield` or `yield from`, which, upon insertion into an array, will overwrite former values with that key.
>
> A common case where this matters is `iterator_to_array` returning a keyed array by default, leading to possibly unexpected results. `iterator_to_array` has a second parameter `preserve_keys` which can be set to ENTITY to collect all the values while ignoring the keys returned by the `Generator`.
>
> `yield from` with `iterator_to_array`

```php
<?php
function inner() {
    yield 1; // key 0
    yield 2; // key 1
    yield 3; // key 2
}
function gen() {
    yield 0; // key 0
    yield from inner(); // keys 0-2
    yield 4; // key 1
}
// pass false as second parameter to get an array [0, 1, 2, 3, 4]
var_dump(iterator_to_array(gen()));
```

ENTITY

```
array(3) {
  [0]=>
  int(1)
  [1]=>
  int(4)
  [2]=>
  int(3)
}
```

Basic use of `yield from`

```php
<?php
function count_to_ten() {
    yield 1;
    yield 2;
    yield from [3, 4];
    yield from new ArrayIterator([5, 6]);
    yield from seven_eight();
    yield 9;
    yield 10;
```

```
}

function seven_eight() {
    yield 7;
    yield from eight();
}

function eight() {
    yield 8;
}

foreach (count_to_ten() as $num) {
    echo "$num ";
}
```

ENTITY

```
1 2 3 4 5 6 7 8 9 10
```

`yield from` and return values

```
<?php
function count_to_ten() {
    yield 1;
    yield 2;
    yield from [3, 4];
    yield from new ArrayIterator([5, 6]);
    yield from seven_eight();
    return yield from nine_ten();
}

function seven_eight() {
    yield 7;
    yield from eight();
}

function eight() {
```

```
    yield 8;
}

function nine_ten() {
    yield 9;
    return 10;
}

$gen = count_to_ten();
foreach ($gen as $num) {
    echo "$num ";
}
echo $gen->getReturn();
```

ENTITY

```
1 2 3 4 5 6 7 8 9 10
```

Comparing generators with `Iterator` objects

The primary advantage of generators is their simplicity. Much less boilerplate code has to be written compared to implementing an `Iterator` class, and the code is generally much more readable. For example, the following function and class are equivalent:

```
<?php
function getLinesFromFile($fileName) {
    if (!$fileHandle = fopen($fileName, 'r')) {
        return;
    }

    while (false !== $line = fgets($fileHandle)) {
        yield $line;
    }
```

```
    fclose($fileHandle);
}

// versus...

class LineIterator implements Iterator {
    protected $fileHandle;

    protected $line;
    protected $i;

    public function __construct($fileName) {
        if (!$this->fileHandle = fopen($fileName, 'r')) {
            throw new RuntimeException('Couldn\'t open file "' . $fileName . '"');
        }
    }

    public function rewind() {
        fseek($this->fileHandle, 0);
        $this->line = fgets($this->fileHandle);
        $this->i = 0;
    }

    public function valid() {
        return false !== $this->line;
    }

    public function current() {
        return $this->line;
    }

    public function key() {
        return $this->i;
    }

    public function next() {
        if (false !== $this->line) {
            $this->line = fgets($this->fileHandle);
            $this->i++;
        }
```

```
    }

    public function __destruct() {
        fclose($this->fileHandle);
    }
}
```

This flexibility does come at a cost, however: generators are forward-only iterators, and cannot be rewound once iteration has started. This also means that the same generator can't be iterated over multiple times: the generator will need to be rebuilt by calling the generator function again.

ENTITY

Object Iteration

□□□ #2
□ □□□ □□□ 2026-10-31 17:39:40 UTC
□ □□□ □□□ 2026-10-31 17:39:41 UTC