NAME

Data::Dump − Pretty printing of data structures

SYNOPSIS

```
use Data::Dump qw(dump);

$str = dump(@list);
@copy_of_list = eval $str;

# or use it for easy debug printout
use Data::Dump; dd localtime;
```

DESCRIPTION

This module provides a few functions that traverse their argument list and return a string containing Perl code that, when `eval`ed, produces a deep copy of the original arguments.

The main feature of the module is that it strives to produce output that is easy to read. Example:

```
@a = (1, [2, 3], {4 => 5});
dump(@a);
```

Produces:

```
"(1, [2, 3], { 4 => 5 })"
```

If you dump just a little data, it is output on a single line. If you dump data that is more complex or there is a lot of it, line breaks are automatically added to keep it easy to read.

The following functions are provided (only the dd* functions are exported by default):

dump(...)
pp(...)
: Returns a string containing a Perl expression. If you pass this string to Perl's built-in **eval()** function it should return a copy of the arguments you passed to **dump()**.

  If you call the function with multiple arguments then the output will be wrapped in parenthesis ''(..., ...)''. If you call the function with a single argument the output will not have the wrapping. If you call the function with a single scalar (non-reference) argument it will just return the scalar quoted if needed, but never break it into multiple lines. If you pass multiple arguments or references to arrays of hashes then the return value might contain line breaks to format it for easier reading. The returned string will never be ''\n'' terminated, even if contains multiple lines. This allows code like this to place the semicolon in the expected place:

  ```
  print '$obj = ', dump($obj), ";\n";
  ```

  If **dump()** is called in void context, then the dump is printed on STDERR and then ''\n'' terminated. You might find this useful for quick debug printouts, but the dd*() functions might be better alternatives for this.

  There is no difference between **dump()** and **pp()**, except that **dump()** shares its name with a not-so-useful perl builtin. Because of this some might want to avoid using that name.

quote(`$string`)
: Returns a quoted version of the provided string.

  It differs from `dump($string)` in that it will quote even numbers and not try to come up with clever expressions that might shorten the output. If a non-scalar argument is provided then it's just stringified instead of traversed.

dd(...)
ddx(...)
: These functions will call **dump()** on their argument and print the result to STDOUT (actually, it's the currently selected output handle, but STDOUT is the default for that).

The difference between them is only that **ddx()** will prefix the lines it prints with ‘‘# ’’ and mark the first line with the file and line number where it was called. This is meant to be useful for debug printouts of state within programs.

dumpf(..., \&filter)

Short hand for calling the **dump_filtered()** function of Data::Dump::Filtered. This works like **dump()**, but the last argument should be a filter callback function. As objects are visited the filter callback is invoked and it can modify how the objects are dumped.

CONFIGURATION

There are a few global variables that can be set to modify the output generated by the dump functions. It’s wise to localize the setting of these.

`$Data::Dump::INDENT`

This holds the string that’s used for indenting multiline data structures. It’s default value is ‘‘ ’’ (two spaces). Set it to "‘‘ to suppress indentation. Setting it to ’’| " makes for nice visuals even if the dump output then fails to be valid Perl.

`$Data::Dump::TRY_BASE64`

How long must a binary string be before we try to use the base64 encoding for the dump output. The default is 50. Set it to 0 to disable base64 dumps.

`$Data::Dump::LINEWIDTH`

This controls how wide the string should before we add a line break. The default is 60.

LIMITATIONS

Code references will be dumped as `sub { ... }`. Thus, `eval`ing them will not reproduce the original routine. The`...`−operator used will also require perl−5.12 or better to be e valed.

If you forget to explicitly import the `dump` function, your code will core dump. That’s because you just called the builtin `dump` function by accident, which intentionally dumps core. Because of this you can also import the same function as `pp`, mnemonic for ‘‘pretty-print’’.

HISTORY

The `Data::Dump` module grew out of frustration with Sarathy’s in-most-cases-excellent `Data::Dumper`. Basic ideas and some code are shared with Sarathy’s module.

The `Data::Dump` module provides a much simpler interface than `Data::Dumper`. No OO interface is available and there are fewer configuration options to worry about. The other benefit is that the dump produced does not try to set any variables. It only returns what is needed to produce a copy of the arguments. This means that`dump("foo")` simply returns `'"foo"'`, and `dump(1..3)` simply returns `'(1, 2, 3)'`.

SEE ALSO

Data::Dump::Filtered, Data::Dump::Trace, Data::Dumper, JSON, Storable

AUTHORS

The `Data::Dump` module is written by Gisle Aas <gisle@aas.no>, based on `Data::Dumper` by Gurusamy Sarathy <gsar@umich.edu>.