Perl and PG

This tutorial on Perl and PG will familiarize you with the basic constructs used most often when writing WebWork questions. It is not comprehensive and does not explain everything. We will often provide links at the bottom of each page in case you want to know more details. You could also use Google to search for more details, or a textbook on Perl. If part of this tutorial is too difficult, has insufficient detail, or starts at too high a level, please let me know by emailing paulZwebZworkATgmail.com (remove the Z's – they're to prevent spam).

PERL stands for Practical Extraction and Report Language. PG stands for Problem Generation, and it is a language built on Perl to write WebWork homework questions. Each PG file is essentially a Perl script that produces output in HTML or PDF (via pdfLaTeX). MathObjects are a recently developed part of the PG language, and will be discussed separately.

Perl Basics

Perl has strings, which are enclosed by single quotes (which are straight up on your keyboard, not slanted to the left) or double quotes, and numbers. If you enclose a number with quotes and perform mathematical operations on it, there will be no problems (Perl will treat the string like a number). The comment character in Perl is `#`. The semicolon `;` ends a line of code in Perl. It is possible to concatenate strings in Perl using a period between them, as in `"War " . "and Peace";` (notice the space between the quote and War).

Online References

- **Perl Language (perl.org)**
- **Perl Documentation (perldoc.perl.org)**
- **Perl tutorial (tizag.com)**

Data Types in Perl

Perl has three principal data types: scalars, arrays, and hashes. You will certainly need to learn how to use scalars, and you should learn how to use arrays. For most WebWork questions you write, you probably won't need hashes.

(1) Scalars are any string or number. Named scalars start with $.

```
$name = "Thomas Jefferson";
$value = -5;
```

(2) Arrays of scalars. Named arrays start with @. Indexing for arrays always starts with zero.

```
@ordinals = ("First","Second","Third");
@values = (11,12,13);
```

For arrays of consecutive numbers or letters, you can use the shortcuts below. Another useful shortcut is the `qw( )` function, which converts a space-separated list into an array.

```
@values = (11..13);
@alphabet = (a..z);
@names = qw(Mike Arnie Davide);
```

(3) Hashes are associative arrays of scalars. Named hashes start with %. An entry in a hash looks like `key => value` and associates a value to each key with notation very similar to function notation.

```
%rankings = (
  First  => "Twins",
  Second => "Yankees"
);
%integers = ( One => 1, Two => 2 );
```

Scalars, arrays, and hashes can also be references to something. References are a bit more advanced and we discuss them later.

Accessing Scalars and Using Array Indices

To access a scalar inside an array or a hash, use

```
$ordinals[0]
$ordinals[-1]
$rankings{Second}
```

to get the values First, Third, and Yankees. These all start with $ because we're accessing a scalar. Notice that the indexing for the array starts with zero, and the last element in the ordinals array can be accessed either with the index 2 or -1. You can get the index for the last element in an array and the total number of elements in an array using

```
$#ordinals
scalar(@ordinals)
```

which return 2 (since the indexing starts with zero) and 3. You can create a new scalar whose value is an entry in an array or hash using

```
$age = $values[1];
$myteam = $rankings{First};
```

which will set `$age = 12;` and `$myteam = "Twins";`

Printing an Array

If you print an array such as `@names` you will get 3, which is not what you expected. If you use `join(", ", @names);` instead, then subsequent elements will be joined by a comma followed by a space and you will get Mike, Arnie, Davide, which is what you want.

Online References

- **Perl data types (cpan.org)**
- **Perl scalar variables (perltutorial.com)**
- **Perl arrays (perltutorial.com)**
- **Perl hashes (perltutorial.com)**

- **Perl strings (tizag.com)**
- **Perl variables (tizag.com)**
- **Perl arrays (tizag.com)**
- **Perl hashes (tizag.com)**

Arithmetic in Perl

The following operations are defined in perl: `+, -, *, /, **, %`, where `2**3` (=8) is exponentiation and `12 % 7` (=5) is modular arithmetic.

The syntax for arithmetic operations in Perl is straightforward with these exceptions.

(1) Always write `*` when multiplying, since juxtaposition does not mean multiply in Perl.

```
5 * 2;  # correct
(5)(2); # incorrect
5 2;    # incorrect
```

(2) Use `**` for exponentiation instead of `^`, since `^` is the shift operator in Perl.

```
5**2; # correct
5^2;  # incorrect
```

(3) When subtracting or negating, use extra space or parentheses.

```
5 - -2;   # correct
5 -(-2);  # correct
5 + -2;   # correct
5 +(-2);  # correct

$a = -2;
5 - $a;   # correct
5 -($a);  # correct
```

which will return the correct values 7, 7, 3, 3, 7, 7. However, if you don't use extra space or parentheses, you run into the problem that `--` is the decrement operator in Perl (e.g., `5--` is the same as `4`).

```
5--2;  # error
```

(4) Be careful with fractional exponents. Perl will evaluate `(-4)**(2/3)` as $e^{(2/3)\ln(-4)}$ which will throw an error since the natural log of -4 is undefined. The correct syntax would be

```
((-4)**2)**(1/3);
```

which will return the value 2.51984209978975 we expect.

From a mathematicians perspective, these problems with Perl arithmetic are rather annoying. In fact, later in the tutorial we will see that a new part of the PG language called MathObjects, which were created by Davide Cervone at Union College, will circumvent the first three problems of Perl arithmetic. MathObjects will not let you raise a negative number to a non-integer power (it will give an error), so you will still have to be careful with fractional exponents.

Online References

- **Perl numbers (perl.org)**
- **Perl operators (perl.org)**
- **Perl numbers (tizag.com)**
- **Perl operators (tizag.com)**

Mathematical Functions in Perl

WebWork automatically loads the following mathematical Perl functions. **We strongly recommend that you use these functions inside of MathObjects whenever possible.** Later in the tutorial there will be an example of how to define a new named function using MathObjects.

- `sqrt()`, square root (there is no cbrt, use fractional exponents instead)
- `sin(), cos(), tan(), sec(), csc(), cot()`, trig functions in radians
- `sinh(), cosh(), tanh(), sech(), csch(), coth()`, hyperbolic trig functions
- `arcsin(), arccos(), arctan(), arcsec(), arccsc(), arccot()`, inverse trig functions in radians
- `asin(), acos(), atan(), asec(), acsc(), acot()`, inverse trig functions in radians
- `exp();`, natural exponential function
- `ln(), log()`, natural logarithm function **Caution: log is the natural logarithm in Perl and MathObjects.**
- `logten()`, base 10 logarithm function
- `abs()`, absolute value
- `sgn()` sign function, either -1, 0, or 1
- `step()` step function `(0 if x < 0, 1 if x >= 0)`
- `fact()` factorial function (defined only for non negative integers)

Online References

- **Perl numbers (tizag.com)**

Relational and Logical Operators in Perl

The following relational and logical operators are defined in Perl. The syntax is different for comparing numbers and strings, and it is important not to mix them up.

(1) == equality for numbers, `eq` equality for strings
(2) != not equal for numbers, `ne` not equal for strings
(3) < less than for numbers, `lt` less than for strings
(4) <= less than or equal for numbers, `le` less than or equal for strings
(5) > greater than for numbers, `gt` greater than for strings
(6) >= greater than or equal for numbers, `ge` greater than or equal for strings
(7) `&&` and for numbers, `and` for strings
(8) `||` or for numbers, `or` for strings

Conditional Statements in Perl

The most commonly used conditional statements when writing WebWork questions are

(1) If-then statements:

```
if ( 5 == 5 ) { $a = 1; }
if ( 5 <= 6 ) { $b = 1; }
if ( "Foo" eq "Foo" ) { }
```

The first two statements are true, so `$a = 1;` and `$b = 1;`. The last statement is true, but no action is taken since it is empty between the curly braces.

(2) If-then-else statements:

```
if ( 5 >= 6 ) {
   $a = 1;
} else {
   $a = 2;
}
```

The first statement is false, so `$a = 2;`

(3) If-then-elsif-then-else statements:

```
if ( 5 >= 6 ) {
   $a = 1;
} elsif ( "Roy" eq "James" ) {
   $a = 2;
} else {
   $a = 3;
}
```

The first two statements are false, so `$a = 3;`

(4) Compound statements:

```
if ( 5 >= 6 || 7 < 10 ) {
   $a = 1;
}
if ( 5 >= 6 && 7 < 10 ) {
   $b = 1;
}
```

The first if statement would set `$a = 1;`, but the second if statement would take no action.

Online References

- **Perl if statements (tizag.com)**
- **Perl operators (tizag.com)**
- **Perl if statements (perltutorial.com)**

For and While Loops in Perl

(1) For loops have the general form

```
for (initial value, test, increment) {
  code;
}
```

In this example, we add up the first four numbers and store their value in `$n`. Notice the recursive assignment `$n = $n ...` is allowed in Perl.

```
$n = 0;
for ($i = 1; $i < 5; $i++) {
   $n = $n + $i;
}
```

(2) Foreach loops run through arrays and have the general form

```
foreach $element @array {
   code;
}
```

and will execute the code for each element of the array. In this example we compute 4 factorial and store it in `$n`. By writing `my $i` (instead of just `$i`) in the foreach loop we declare the variable `$i` is local (it has limited scope). This means that outside of the foreach loop the variable `$i` always takes the value 75, while inside the foreach loop the variable `$i` will take the values in the array `(1..4)`, and the values outside and inside the foreach loop never interfere with each other.

```
$i = 75;
$n = 1;
foreach my $i (1..4) {
   $n = $n * $i;
}
```

Foreach loops can also be used to fill arrays with values.

```
@evens = ();
foreach my $i (0..10) {
   $evens[$i] = 2*$i;
}
```

(3) While loops have the general form

```
while (condition) {
  code;
}
```

and will continue to execute code as long as the condition remains true. In this example, we increment `$i` by +1 so long as it is not equal to 5. We could have used < instead of != and gotten the same end result `$i = 5;`.

```
$i = 0;
while ($i != 5) { $i++; }
```

Online References

- **Perl for loops (perltutorial.com)**
- **Perl while loops (perltutorial.com)**
- **Perl for loops (tizag.com)**
- **Perl while loops (tizag.com)**

Subroutines in Perl

Named subroutines (or procedures or methods) are blocks of code that can take scalar inputs and return outputs. They have the general form `sub name { code; }`.

(1) In this example we use a subroutine to define a function.

```
sub fx { my $t=shift(); return 4*cos($t); }
```

The shift operator grabs the input to the subroutine, which is then stored in the local scalar variable `$t`. This

named subroutine can be used later, for example, to create an array `@xcoord` of x-coordinates of points on the radius 4 circle centered at the origin.

```
foreach my $i (0..6) {
   $xcoord[$i] = fx($i);
}
```

(2) In this example, we create a subroutine that returns the max of two numerical inputs.

```
sub max {
    $a = shift();
    $b = shift();
    if ($a >= $b) {
       return $a;
    } else {
       return $b;
    }
}
max(-1,5);
```

(3) By default, the inputs to a subroutine are stored in the local array `@_` and the scalars in this array can be accessed via `$_[0]`, `$_[1]`, etc. We could rewrite our max subroutine as follows.

```
sub max {
    if ($_[0] >= $_[1]) {
       return $_[0];
    } else {
       return $_[1];
    }
}
max(-1,5);
```

Online References

- **Perl subroutines (perltutorial.com)**
- **Perl sort (perl.org)**
- **Perl references (perl.org)**

References in Perl

Recall that arrays and hashes in Perl are arrays and associative arrays of *scalars.* This means that Perl does not have arrays of arrays (i.e., matrices). References were introduced in Perl 5 to make up for this shortcoming. A reference is a scalar that can refer to an entire array, and if you have a reference to an array you can recover the array from it. Since a reference is a scalar, you can have an array of references, which is every bit as useful as an array of arrays.

(1) If you put a backslash in front of an array or hash, yet get a reference to it.

```
@array = (11..15);
%hash = ( Jan => 1, Feb => 2 );
$arrayref = \@array;
$hashref  = \%hash;
```

(2) To recover an array or a hash from a reference, use `@{$ref}` or `%{$ref}`

```
@newarray = @{$arrayref};
%newhash  = %{$hashref};
```

(3) You can use the array `@{$ref}` and the hash `%{$ref}` just like any other array or hash. For example, we access the scalars in these arrays.

```
${$arrayref}[1];    # 12
$newarray[1];       # 12
${$hashref}{"Feb"}; # 2
$newhash{"Feb"};    # 2
```

(4) You can create unnamed (or anonymous) arrays `[ items ]` and hashes `{ items }` that return references. We create some of these below and assign them to scalar variables.

```
$aref = [11,12,13];
$href = { Mar => 3, Apr => 4 };
```

(5) We can use the unnamed (or anonymous) arrays to create a named array of unnamed arrays (i.e., an array of references).

```
@matrix = (
[ "a00", "a01", "a02" ],
[ "a10", "a11", "a12" ],
[ "a20", "a21", "a22" ]
);
```

(6) We can access the scalars in this matrix using `$matrix[rowindex][columnindex]`. Remember: the indexing always starts with zero.

```
$matrix[1][2]; # element "a12"
```

(7) References can also be used to pass arrays to subroutines. For example, suppose you want to create a subroutine that takes in two references to arrays of data and processes them somehow.

```
# arrays of data
@xdata = (10,23,31);
@ydata = (95,77,84);

sub data_processor {
    # read references
    my $xref = shift;
    my $yref = shift;
    # convert references to arrays
    my @x = @{$xref};
    my @y = @{$yref};
    # now process arrays @x and @y
}

# inputs are references to arrays
data_processor( \@xdata, \@ydata );
```

Online References

- **Perl references (perl.org)**

PG document structure

A PG file uses Perl to produce a document that can be output in HTML or PDF format. The general structure of a PG file is much like the structure of HTML source code or a TeX source file that becomes a PDF file when compiled by pdfLaTeX. PG files typically have the following structure.

(1) Tagging information: With over 20,000 PG files in the National Problem Library, every PG file should begin with metadata can be used to index it in the library. The tags `DBsubject()`, `DBchapter()`, and `DBsection()` are necessary to put the PG file into the hierarchical framework of the database. The best way to find where your PG file fits in the database is to open the library browser and find the appropriate subject, chapter, and section. I like to view the source code of a problem that has the right subject, chapter, and section, and just copy and paste this information into my PG file. (Note: If you're copying and pasting from a PDF to a PG file, the single quotes will be slanted, so you'll have to replace them by non-slanted single quotes. Ahh, the quirks of TeX.)

```
## DESCRIPTION
## PG document structure
## ENDDESCRIPTION

## KEYWORDS('sample', 'WeBWorK')

## DBsubject('WeBWorK')
## DBchapter('WeBWorK Tutorial')
## DBsection('Fort Lewis Tutorial 2011')
## Date('01/30/2011')
## Author('Paul Pearson')
## Institution('Fort Lewis College')
## TitleText1('')
## EditionText1('')
## AuthorText1('')
## Section1('')
## Problem1('')
```

(2) Initialization: This is where we load macros and begin the document. The macros `PGstandard.pl` and `MathObjects.pl` should always be loaded. There are many macros that can be loaded, so rather than trying to describe them all here, we will let you look at which macros are loaded in the template examples later in this tutorial.

```
##########################
#  Initialization

DOCUMENT();

loadMacros(
"PGstandard.pl",
"MathObjects.pl",
);

# TEXT(beginproblem());

$refreshCachedImages = 1;
```

(3) Setup: This is where we use Perl, PG, and MathObjects to write the inner workings of the problem. We use MathObjects to set the context (Numeric, Vector, etc.), and define a MathObject using Compute.

```
##########################
#  Setup

Context("Numeric");

$r = random(3,9,1);
$answer = Compute("pi * $r^2");
```

(4) Main Text: Here we enter PG's text mode, between `BEGIN_TEXT` and `END_TEXT`, and display the question for the student. The `BEGIN_TEXT` and `END_TEXT` commands must be at the beginning of the line (no leading spaces) and also be the only thing on their line. While in text mode, we can access Perl mode via escaped curly braces \{ Perl code here \}, and TeX inline mode via escaped rounded parentheses \(TeX code here \), and TeX display mode (centered) via escape square brackets \[TeX code here \]. The code `Context()->texStrings;` and `Context()->normalStrings;` outside of the text block will ensure that any MathObjects that are used in TeX mode will be beautifully typeset. The code `\{ ans_rule(10) \}` produces an answer blank 10 characters wide.

```
##########################
#  Main Text

Context()->texStrings;
BEGIN_TEXT
What is the area of a circle of radius
\($r\)?
$BR
$BR
Area = \{ ans_rule(10) \}
END_TEXT
Context()->normalStrings;
```

(5) Answer Evaluation: This is where we use MathObjects to check whether the student's answer is correct. We defined the MathObject `$answer` above in the Setup section, and now we will call the method `cmp()` on it using

Perl's syntax for object-oriented programming (the arrow). The subroutine `ANS( )` will record the result in the gradebook database.

```
######################
#  Answer Evaluation

$showPartialCorrectAnswers = 1;

ANS( $answer->cmp() );
```

(6) Solution and End Document: The solution is optional, but ending the document is mandatory (don't forget it!). With older versions of WebWork, it may be necessary to use `SOLUTION(EV3(<<'END_SOLUTION'));` instead of `BEGIN_SOLUTION`.

```
######################
#  Solution

Context()->texStrings;
BEGIN_SOLUTION
The formula for the area of
a circle of radius \(r\)
is \(\pi r^2\),
so the answer is
\($answer\).
END_SOLUTION
Context()->normalStrings;

ENDDOCUMENT();
```

Online References

- **Tagging WebWork Problems**
- **Standard PG macros (POD documentation)**
- **Standard PG macros (source)**
- **Submitted PG macros (source)**

Random Numbers in PG

The macro file `PGstandard.pl` automatically loads `PGbasicmacros.pl`, which provides several random number generating utilities. The list random generator chooses one item from the comma separated list of numbers.

```
random(low,high,increment);
non_zero_random(low,high,increment);
list_random(list of numbers);
```

If you want random values between 0.2 and 0.5 with increment 0.1, you should use the list random option to avoid any strange computer rounding errors (computers are binary, so they don't like tenths!). For example:

```
$a = list_random(0.2,0.3,0.4,0.5);
$b = random(0.2,0.5,0.1); # might return 0.299999999
```

Common Randomization Recipes

(1) Two distinct random integers:

```
$a = random(2,9,1);
do { $b = random(2,9,1); } until ( $b != $a);
```

(2) Three distinct random integers:

```
$a = random(2,9,1);
do { $b = random(2,9,1);
} until ( $b != $a );
do { $c = random(2,9,1);
} until ( ($c != $a) && ($c != $b) );
```

(3) Generating an array of random integers:

```
@a = ();
foreach my $i (0..8) {
   $a[$i] = random(2,9,1);
}
# access these using $a[0], $a[1], etc.
```

(4) Controlling the size and the sign:

```
$s = random(-1,1,2);
$a = random(4,14,1);
$b = $s * $a;
```

Shared Randomization Across Files

If you have two different PG files and you want to use them in the same homework set and have the same randomization, you need to set the randomization seed to be the same in each file. To do this, put the following code into both files before any calls to random number generators. The SRAND function sets the randomization seed, and psvn stands for problem set version number.

```
SRAND($psvn);
```

Online References

- **Standard PG macros (POD documentation)**
- **Standard PG macros (source)**

Special Characters and Commands in PG

(1) Escape characters: Both TeX and Perl have backslash their escape character (think of all the commands that start with backslash like `\newline` in LaTeX and `\n` in Perl). Since PG is built on both TeX and Perl, this creates a conflict — in PG, which language (TeX or Perl) should get to use backslash as its escape character? To resolve this conflict, PG reserves backslash as the escape character for all TeX commands. To get the Perl escape code to work in PG, use two tildes ∼∼ instead of backslash. When the PG file is executed, the two tildes (in PG mode) will automatically be remapped to backslash (in Perl mode). So, if you have a reference to an array in PG, you should use ∼∼`@array` instead of `\@array`.

(2) A text block in a PG file is enclosed by `BEGIN_TEXT` and `END_TEXT`, `BEGIN_HINT END_HINT`, or `BEGIN_SOLUTION END_SOLUTION`. Everything in a text block is in a new mode determined by PG, and it is possible to temporarily switch to TeX inline mode

using \(\), TeX display mode using \[\], or Perl mode using \{ \}. Inside a text block, you will **not** be able to access some special characters like $, %, ^, _ simply by typing them, so PG has special commands for them.

```
$DOLLAR  produces \$
$PERCENT produces %
$CARET   produces ^
$US      produces _
```

In a text block, you may want to break a line, create a new paragraph, center text, or make text bold, italic, underlined, or quoted. The commands for these things are different in HTML and TeX, so PG provides you with commands that work properly in both HTML and TeX. Notice that, for example, we have written `${BBOLD}Text made bold${EBOLD}` with extra curly braces around `${BBOLD}` and `${EBOLD}` to keep them from running together with the text they enclose. However, since the centered text is on a separate line from `$BCENTER` and `$ECENTER`, there is no chance of things running together and we omit the extra curly braces.

```
$BR  produces a line break
$PAR produces a paragraph break

$BCENTER
Text made centered
$ECENTER

${BBOLD}Text made bold${EBOLD}
${BITALIC}Text made italic${EITALIC}
${BUL}Text underlined${EUL}
${LQ}Quoted text with curly TeX quotes${RQ}
```

If you want inequalities or curly braces, use TeX's math mode.

```
\(<\)       less than
\(>\)       greater than
\(\leq\)    less than or equal
\(\geq\)    greater than or equal
\(\lbrace\) left curly brace
\(\rbrace\) right curly brace
```

The commands just discussed are defined in `PGbasicmacros.pl`.

Online References

- **Standard PG macros (POD documentation)**
- **Standard PG macros (source)**

Auxiliary Functions in PG

The macro file PGauxiliaryFunctions.pl, which is automatically loaded by PGstandard.pl, contains many useful numerical recipies.

(1) The max function `max(-1,4,2*pi,5);` will return 6.28319.

(2) The min function `min(10,2*pi,-1,5);` will return -1.

(3) The greatest common factor (or divisor) function `gcf(4,6);` or `gcd(4,6);` will return 2. For perl code that gives an extended gcd function `xgcd(a,b) = d = ax`by+, see the online references below.

(4) The least common multiple function `lcm(4,6);` will return 12.

(5) The ceiling and floor functions `ceil(-3.4);` and `floor(-3.4);` will return -3 and -4.

(6) The prime test function `isPrime(4);` and `isPrime(5);` will return 0 and 1.

(7) The signum (or sign) function `sgn(-pi);` and `sgn(0);` and `sgn(6)` will return -1, 0, and 1.

(8) The Heaviside step function, which is 1 when the input is positive and zero otherwise, is `step(-0.1);` and `step(0);` and `step(0.1);` which returns 0, 0, and 1.

(9) The integer rounding function `round(1.49999);` and `round(1.5);` and `round(-1.5);` and `round(-1.49999);` will return 1, 2, -2, -1.

Online References

- **Standard PG macros (POD documentation)**
- **Standard PG macros (source)**
- **Extendend gcd function**

Introduction to MathObjects

Perl has very limited data types: scalars (i.e., numbers and strings), and arrays and hashes of scalars. To Perl, a string like `"sin(x^2+6)"` is just a string, not a function. MathObjects were created by Davide Cervone at Union College to address this and other problems. MathObjects are a set of formal objects introduced to make manipulation of mathematical objects in WeBWorK problems more intuitive. For instance, `Formula("sin(x^2+6)")` is a MathObject that takes the Perl string `"sin(x^2+6)"` and makes it a function that can be evaluated, differentiated, can produce a TeX representation, can produce a Perl function (i.e., a subroutine that returns the value of this function), etc.

MathObjects are useful and powerful because they are much more than just a number or a string — they know what context they live in, mathematical operations can be done with them, and they have methods defined on them.

(1) MathObjects know what context they live in:

```
Context("Numeric");
$f = Formula("sin(x^2/3)");
$a = Real("sqrt(pi/2)");
```

```
Context("Vector");
$v = Vector("<1,2,3>");

Context("Inequalities");
$domain = Inequality("-16 < x < 9");
```

In fact, we could replace `Formula`, `Real`, `Vector`, and `Inequality` all by `Compute`, in which case MathObjects would discern the type of object from the context. `Compute` is usually the preferred way to convert a string to a MathObject. It preserves the original string and uses it to display the correct answer expected of the student in the most useful form. The angle brackets in the vector and the inequality are interpreted differently (and correctly) because these MathObjects know which context they live in.

(2) Mathematical operations can be done with MathObjects:

```
Context("Numeric");
$f = Compute("sin(x^2/3)");
$a = Compute("sqrt(pi/2)");
$g = $a * $f;
```

Note that `$g` is a MathObject, as it is the product of two MathObjects that live in the same context. Mathematical operations cannot be done with MathObjects that live in different contexts.

(3) MathObjects have methods defined on them:

```
Context("Numeric");
$f = Compute("sin(x^2/3)");
$a = Compute("sqrt(pi/2)");

# evaluate f(a)
$b = $f->eval(x=>$a);

# partial derivative df/dx
$fx = $f->D('x');

# basic algebraic simplification to x+4
$c = -4;
$g = Compute("x - $c")->reduce();

# get a string of TeX code for f
# returns "\sin\!\left(\frac{x^{2}}{3}\right)"
$h = $f->TeX();
```

We have called the evaluate `->eval()`, partial differentiation `->D('x')`, very basic algebraic simplification `->reduce()`, and TeX string `->TeX()` methods on MathObjects.

Why Use MathObjects?

(1) MathObjects convert ordinary strings into a rich variety of data types such as points, vectors, matrices, real numbers, complex numbers, inequalities, etc. (Before MathObjects, Perl strings were the best game in town.)

(2) You can use a single MathObject, such as a formula, to produce numeric values, other formulas, TeX output, and answer strings. This avoids having to type the function a multitude of different ways, which would make maintaining the problem harder. (Before MathObjects, you often had to produce each of these separately and manually.)

(3) The answer checkers for MathObjects are better at giving students feedback on syntax errors and are more versatile. (Before MathObjects, very little feedback was given.)

(4) The basic syntax `$f->cmp()` for comparing a student's answer to the correct answer `$f` is the same no matter what type of MathObject `$f` is. (Before MathObjects, you had to know the name of the answer checker you wanted to apply to a Perl string.)

Online References

- **Talk on MathObjects**
- **MathObjects documentation**
- **Introduction to MathObjects**
- **Using MathObjects**
- **MathObjects Answer Checkers**
- **MathObjects README**

Contexts in MathObjects

Contexts in MathObjects can be used to restrict the type of answer students are allowed to enter. For example, you may want to require your students to expand a factored polynomial and combine all like terms, in which case you could use the LimitedPolynomial context. The contexts I use most often are listed below. All of these context require loading the `PGstandard.pl` and `MathObjects.pl` macros, in addition to any specialized macros listed. For a detailed list of all available contexts, see **Specialized Contexts**.

(1) `Context("Numeric");`
Allow numbers and formulas to be entered.
Additional macros required: none

(2) `Context("Fraction-NoDecimals");` and `Context("LimitedProperFraction");`
Require students to enter fractions, or fully simplified fractions.
Additional macros required: `contextFraction.pl`.

(3) `Context("Inequalities");` and `Context("Inequalities-Onl`
Allow intervals to be entered as intervals or inequalities, or require them only to be entered as inequalities.
Additional macros required: `contextInequalities.pl`.

(4) `Context("Point");`
Allow points to be entered.
Additional macros required: none

(5) `Context("Vector");` and `Context("Vector2D");`
Allows vectors to be entered.
Additional macros required: `parserVectorUtils.pl`

(6) `Context("LimitedPolynomial-Strict");`
Allows only fully simplified polynomials as answers.
Additional macros required: `contextLimitedPolynomial.pl`

(7) `Context("LimitedPowers");`
Restrict the base or power allowed in exponentials.
Additional macros required: `contextLimitedPowers.pl`

(8) `Context("PolynomialFactors");`
Allow only entry of polynomials, and their products and powers.
Additional macros required: `contextPolynomialFactors.pl`

We give an example of how to set up the Limited Proper Fraction context. Setting up other contexts is similar.

```
loadMacros(
"PGstandard.pl",
"MathObjects.pl",
"contextFraction.pl",
);

Context("LimitedProperFraction");
```

Online References

- **Specialized contexts**
- **Wiki docs on contexts**
- **Specialized parsers**
- **POD documentation**
- **PG macros**

Using Contexts in MathObjects

A MathObjects context controls the names and values of variables and constants, the functions that are available, the mathematical operations that are available, the way parentheses and angle brackets are interpreted, the strings that are allowed, and various other settings involving tolerances and display formats. We list some of the more commonly used features of contexts below.

(1) Specifying which variables are defined in the context. By default, the variable x is in the Numeric context. In the first example, we add the variable y to the context, while the second example sets t as the only variable allowed (if a student enters a function of x, they will get an error message). Whenever `Context()` is typed, it refers to the current context.

```
Context("Numeric");
Context()->variables->add(y=>"Real");
Context()->variables->are(t=>"Real");
```

(2) Adding strings (words) to the context. By default, the Numeric context already has the case-insensitive strings "NONE", "DNE", "INF", "INFINITY", and perhaps some others. We give an example of adding the case-insensitive strings "True" and "T" to the context, and letting "T" be an alias for "True". If you have a lot of strings to add to the context, use the **auto strings parser**

```
Context("Numeric");
Context()->strings->add(
  True => {},
  T => {alias=>"True"}
);
```

(3) Disabling operations and functions. You can require students to enter their answers in a particular form by disabling functions and operations. We give an example where we do not allow exponentiation, the natural exponential and logarithm, and aany trigonometric functions. For more details, see **disabling functions and operations**

```
Context("Numeric");
Context()->operators->undefine("^","**");
Context()->functions->disable("exp","log");
Context()->functions->disable("Trig");
```

(4) Adding named functions to the context. Suppose we want to add a base two logarithm to the context, so that students could enter `log2( )`. For more details, see **adding functions to the context**

```
loadMacros(
"PGstandard.pl",
"MathObjects.pl",
"parserFunction.pl",
);

Context("Numeric");
parserFunction("log2(x)"=>"log(x)/log(2)");
```

(5) Specifying the values the variables may take (i.e., domain of function evaluation). For formulas, WebWork checks student answers against the correct answer by pointwise comparison at many points. The default is that each variable is evaluated on the interval [-1,1]. This default should be changed if the formula is not defined on the part of the interval [-1,1] or if the function values are not between 10^{-4} and 10^{6}, e.g., if there is a vertical asymptote or the formula is close to being identically zero. It is also possible to specify the domain

for each individual function being compared to the student's answers, as we shall see later.

```
Context("Numeric");
Context()->variables->set(x=>{limits=>[2,5]});
```

(6) Specifying the numerical tolerances for answers being accepted as correct. The student's answer may differ slightly from the correct answer and still be marked correct. The default tolerance is a relative tolerance of 0.1 percent of the correct answer, which may allow many incorrect answers to be marked correct when the correct answer is very large. It is also possible to specify an absolute tolerance, in which case the student's answer must be correct to however many decimal places you specify. The numerical tolerance may also be specified at the time of answer evaluation.

```
Context("Numeric");
Context()->flags->set(
  tolerance=>0.0001,
  tolType=>relative
);

Context()->flags->set(
  tolerance=>0.0000001,
  tolType=>absolute
);
```

Online References

- **Variables in the context**
- **Adding strings to the context**
- **Parser Auto Strings**
- **Disabling functions and operators**
- **Adding functions to the context**
- **Formula test points**
- **Numerical tolerance**
- **Context flags**

Creating MathObjects

Creating MathObjects is easy. MathObjects has a number of predefined constants such as `pi`, `e`, `Infinity`. The following MathObjects can be created by using `loadMacros("PGstandard.pl","MathObjects.pl");` and `Context("Numeric");` unless specified otherwise. If there is more than one correct answer to a question (or even if there might be more than one answer to a question), you should create a list of answers. If you create a MathObject from a (Perl) string, use double quotes `"1+$a x^2"` so that the value for `$a` gets substituted (single quotes `' '` would prevent this from happening).

- `$a = Real(25.238);`
- `$f = Formula("1 + $a x^2");`
- `$p = Point(5,-2,3);`
- `$I = Interval("(2,3]");`
- `$U = Union("(2,3]U[4,5)");`
- `$s = String("DNE");`
- `$M = Matrix([1,1,1],[0,1,1],[0,0,1]);`
- `$I = Infinity;`
- `$L = List(5,"1+x");`
  `$L = List("5,1+x");`
- `$S = Set(-5,0,9);`
  `$S = Set("{-5,0,9}");`
  `$S = Set(); # empty set`
  `$S = Set("{}"); # empty set`
- `Context("Vector");`
  `$v = Vector(2,4,-1);`
  `$v = Vector("<2,4,-1>");`
  `$v = Vector("2i + 4j - k");`
- `Context("Complex");`
  `$z = Complex(1,sqrt(5));`
  `$z = Complex("1+5i");`
- `loadMacros("contextInequalities.pl");`
  `Context("Inequalities"); \leavevmode\\\relax`
  `$I = Inequality("2 < x <= 3");`

You can set the properties of a MathObject after you create it. We give an example of a function that is best tested on small positive integers since `(-1)^n` may give errors otherwise.

```
Context("Numeric")->variables->are(n=>"Real");
$g = Formula("(-1)^n / n!");
$g->{test_points} = [[2],[3],[4],[5],[6]];
```

In another example, we set the limits (domain) for a particular function rather than for the whole context.

```
Context("Numeric");
$f = Formula("sqrt(x-1)");
$f->{limits} = [2,5];
```

Online References

- **Test points for function evaluation**

Methods Defined on MathObjects

All MathObjects have methods defined on them, such as `cmp`, `perl`, `perlFunction`, `value`, `TeX`, `string`, `stringify`, and `getFlag`. If the MathObject is a Formula, it also has the methods `eval`, `reduce`, `substitute`, and the differentiation operator `D` defined on them. We discuss the most commonly used of these methods.

- The method `cmp` is used when checking student's answers. For example,

```
$answer1 = Compute("tan(x)");
$answer2 = List(Formula("5x"),Formula("3"));

ANS( $answer1->cmp() );
ANS( $answer2->cmp() );
```

- The method `TeX` produces TeX code from a MathObject. For example,

```
$f = Compute("sin(x)/x");
```

```
BEGIN_TEXT
\(\displaystyle $f->TeX\)
END_TEXT
```

where `$f->TeX` produces the TeX code `\frac{\sin(x)}{x}`. To apply the TeX method automatically to every MathObject occurring inside a text block, you can change the context to `texStrings` to interpret all MathObject strings as TeX strings, but don't forget to change the context back to `normalStrings`.

```
Context()->texStrings;
BEGIN_TEXT
\(\displaystyle $f->TeX\)
END_TEXT
Context()->normalStrings;
```

- The method `reduce` method will perform very elementary simplification of formulas, such as changing `++` to `+`, `+-` or `-+` to `-`, and `--` to `-`. For example,

```
$a = -1;
$f = Formula("x + $a y - $a z")->reduce;
```

will make it the same as `$f = Formula("x-y+z")`.

- Calling the method `eval` on a formula produces a numerical value (a Real).

```
$f = Formula("sin(x)");
$g = $f->eval(x=>pi/2);
```

will give `$g` the value 1.

- Calling the method `substitute` on a formula produces a MathObject Formula (not a Real) with its parse string intact (which may give more insightful correct answer hints from the answer evaluator).

```
$f = Formula("sin(x)");
$g = $f->substitute(x=>pi/2);
```

will give `$g` the value `Formula("sin(pi/2)")`.

- Calling the method `D` on a formula produces the derivative as a MathObject Formula.

```
$f = Formula("sin(x)");
$g = $f->D('x');
```

will set `$g = Formula("cos(x)")`.

Online References

- **Methods defined on MathObjects**

Get Started Writing WeBWorK Questions

Congratulations on finishing the tutorial! I hope that it has provided you with enough of the basics of Perl, PG, and MathObjects so that you can look at source code, understand what it is doing, and write some of your own questions. I encourage you to start looking at the examples and the documentation in the links below. In particular, the complete templates by subject area and the code snippets of problem techniques are invaluable resources that are (for the most part) up to date and are good models for your own code. We have also included links to other common resources that may be more advanced.

Online References

- **Complete templates by subject area**
- **Code snippets of problem techniques**
- **MathObjects documentation**
- **POD documentation**