Functions

A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. It may or may not have a name. It may or may not consume incoming information. It may or may not produce outgoing information. It represents a type of control flow, where the execution of the program proceeds to another point in the source code.

Functions are a prime mechanism for abstraction, encapsulation, and re-use in Perl 5; many other mechanisms build on the idea of the function.

Functions

Use the `sub` keyword to declare a function:

```
B<sub> greet_me  { ... }
```

Now `greet_me()` is available for invocation anywhere else within the program, provided that the symbol--the function's name--is visible.

You do not have to *define* a function at the point you declare it. You may use a *forward declaration* to tell Perl that you intend for the function to exist, then delay its definition:

```
sub greet_sun;
```

You do not have to declare Perl 5 functions before you use them, except in the special case where they modify *how* the parser parses them. See *attributes*.

Invoking Functions

To invoke a function, mention its name and pass an optional list of arguments:

```
greet_me( 'Jack', 'Brad' );
greet_me( 'Snowy' );
greet_me();
```

You can *often* omit parameter-grouping parentheses if your program runs correctly with the `strict` pragma enabled, but they provide clarity to the parser and, more importantly, the reader.

You can, of course, pass multiple *types* of arguments to a function:

```
greet_me( $name );
greet_me( @authors );
greet_me( %editors );
```

... though see the *references* section for more details.

Function Parameters

Inside the function, all parameters exist in a single array, `@_`. If `$_` corresponds to the English word *it*, `@_` corresponds to the word *them*. Perl *flattens* all incoming parameters into a single list. The function itself either must unpack all parameters into any variables it wishes to use or operate on `@_` directly:

```
sub greet_one
{
    B<my ($name) = @_>;
    say "Hello, $name!";
}

sub greet_all
{
    say "Hello, B<$_!" for @_>;
}
```

@_ behaves as does any other array in Perl. You may refer to individual elements by index:

```
sub greet_one_indexed
{
    B<my $name = $_[0]>;
    say "Hello, $name!";

    # or, less clear
    say "Hello, $_[0]!";
}
```

You may also `shift`, `unshift`, `push`, `pop`, `splice`, and use list slices on `@_`. Inside a function, the `shift` and `pop` operators operate on `@_` implicitly in the same way that they operate on `@ARGV` outside of any function:

```
sub greet_one_shift
{
    B<my $name = shift>;
    say "Hello, $name!";
}
```

While writing `shift @_` may seem clearer initially, taking advantage of the implicit operand to `shift` is idiomatic in Perl 5.

Take care that assigning a scalar parameter from `@_` requires `shift`, indexed access to `@_`, or lvalue list context parentheses. Otherwise, Perl 5 will happily evaluate `@_` in scalar context for you and assign the number of parameters passed:

```
sub bad_greet_one
{
    B<my $name = @_>;  # buggy
    say "Hello, $name; you're looking quite numeric today!"
}
```

List assignment of multiple parameters is often clearer than multiple lines of `shift`. Compare:

```
sub calculate_value
{
    # multiple shifts
    my $left_value  = shift;
    my $operation   = shift;
    my $right_value = shift;
    ...
}
```

... to:

```
sub calculate_value
{
    B<my ($left_value, $operation, $right_value) = @_;>
    ...
}
```

Occasionally it's necessary to extract a few parameters from `@_` and pass the rest to another function:

```
sub delegated_method
{
    my $self = shift;
    say 'Calling delegated_method()'

    $self->delegate->delegated_method( @_ );
}
```

The dominant practice seems to be to use `shift` only when your function must access a single parameter and list assignment when accessing multiple parameters.

See the `signatures`, `Method::Signatures`, and `MooseX::Method::Signatures` modules on the CPAN for declarative parameter handling.

Flattening

The flattening of parameters into `@_` happens on the caller side. Passing a hash as an argument produces a list of key/value pairs:

```
sub show_pets
{
    my %pets = @_;
    while (my ($name, $type) = each %pets)
    {
        say "$name is a $type";
    }
}

my %pet_names_and_types = (
    Lucky   = > 'dog',
    Rodney  = > 'dog',
    Tuxedo  = > 'cat',
    Petunia = > 'cat',
);

show_pets( %pet_names_and_types );
```

The `show_pets()` function works because the `%pet_names_and_types` hash flattens into the list `'Lucky', 'dog', 'Rodney', 'dog', 'Tuxedo', 'cat', 'Petunia', 'cat'`. The hash assignment inside the function `show_pets()` works essentially as the more explicit assignment to `%pet_names_and_types` does.

This is often useful, but you must be clear about your intentions if you pass some arguments as scalars and others as flattened lists. If you wish to make a `show_pets_of_type()` function, where one parameter is the *type* of pet to display, you must pass that type as the *first* parameter (or use `pop` to remove it from the end of `@_`):

```
sub show_pets_by_type
{
    B<my ($type, %pets) = @_>;
```

```
    while (my ($name, $species) = each %pets)
    {
        B<next unless $species eq $type;>
        say "$name is a $species";
    }
}

my %pet_names_and_types = (
    Lucky   = > 'dog',
    Rodney  = > 'dog',
    Tuxedo  = > 'cat',
    Petunia = > 'cat',
);

show_pets_by_type( 'dog',   %pet_names_and_types );
show_pets_by_type( 'cat',   %pet_names_and_types );
show_pets_by_type( 'moose', %pet_names_and_types );
```

Slurping

As with any lvalue assignment to an aggregate, assigning to `%pets` within the function *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would attempt to assign an odd number of elements to the hash and would produce a warning. You *could* work around that:

```
sub show_pets_by_type
{
    B<my $type = pop;>
    B<my %pets = @_;>

    ...
}
```

... at the expense of some clarity. The same principle applies when assigning to an array as a parameter, of course. See *references* for ways to avoid flattening and slurping when passing aggregate parameters.

Aliasing

One useful feature of `@_` can surprise the unwary: it contains aliases to the passed-in parameters, until you unpack `@_` into its own variables. This behavior is easiest to demonstrate with an example:

```
sub modify_name
{
    $_[0] = reverse $_[0];
}

my $name = 'Orange';
modify_name( $name );
say $name;

# prints C<egnarO>
```

If you modify an element of `@_` directly, you will modify the original parameter directly. Be cautious.

Namespaces

Every function lives in a namespace. Functions in an undeclared namespace--that is, functions not declared after an explicit `package ...` statement--live in the `main` namespace. You may specify a function's namespace outside of the current package at the point of declaration:

```
sub B<Extensions::Math::>add {
    ...
}
```

Any prefix on the function's name which follows the package naming format creates the function and inserts the function into the appropriate namespace, but not the current namespace. Because Perl 5 packages are open for modification at any point, you may do this even if the namespace does not yet exist, or if you have already declared functions in that namespace.

You may only declare one function of the same name per namespace, lest Perl 5 warn you about about subroutine redefinition. If you're certain you want to *replace* an existing function, disable this warning with `no warnings 'redefine'`.

You may call functions in other namespaces by using their fully-qualified names:

```
package main;

Extensions::Math::add( $scalar, $vector );
```

Functions in namespaces are *visible* outside of those namespaces in the sense that you can refer to them directly, but they are only *callable* by their short names from within the namespace in which they are declared--unless you have somehow made them available to the current namespace through the processes of importing and exporting (*exporting*).

Importing

When loading a module with the `use` keyword (*modules*), Perl automatically calls a method named `import()` on the provided package name. Modules with procedural interfaces can provide their own `import()` which makes some or all defined symbols available in the calling package's namespace. Any arguments after the name of the module in the `use` statement get passed to the module's `import()` method. Thus:

```
use strict;
```

... loads the *strict.pm* module and calls `strict->import()` with no arguments, while:

```
use strict 'refs';
use strict qw( subs vars );
```

... loads the *strict.pm* module, calls `strict->import( 'refs' )`, then calls `strict->import( 'subs', vars' )`.

You may call a module's `import()` method directly. The previous code example is equivalent to:

```
BEGIN
{
    require strict;
    strict->import( 'refs' );
    strict->import( qw( subs vars ) );
}
```

Be aware that the `use` keyword adds an implicit `BEGIN` block around these statements so that the `import()` call happens *immediately* after the parser has compiled the entire statement. This ensures that any imported symbols are visible when compiling the rest of the program. Otherwise, any functions imported from other modules but not declared in the current file would look like undeclared barewords and `strict` would complain.

Reporting Errors

Within a function, you can get information about the context of the call with the `caller` operator. If passed no arguments, it returns a three element list containing the name of the calling package, the name of the file containing the call, and the line number of the package on which the call occurred:

```
package main;

main();

sub main
{
    show_call_information();
}

sub show_call_information
{
    my ($package, $file, $line) = caller();
    say "Called from $package in $file at $line";
}
```

You may pass a single, optional integer argument to `caller()`. If provided, Perl will look back through the caller of the caller of the caller that many times and provide information about that particular call. In other words, if `show_call_information()` used `caller(0)`, it would receive information about the call from `main()`. If it used `caller(1)`, it would receive information about the call from the start of the program.

While providing this optional parameter lets you inspect the callers of callers, it also provides more return values, including the name of the function and the context of the call:

```
sub show_call_information
{
    my ($package, $file, $lineB<, $func>) = caller(B<0>);
    say "Called B<$func> from $package in $file at $line";
}
```

The standard `Carp` module uses this technique to great effect for reporting errors and throwing warnings in functions. You may see `croak()` used to throw exceptions in place of `die` in library code. `croak()` throws an exception reported from the file and line number of its caller. The `carp()` function reports a warning from the file and line number of its caller.

This behavior is most useful when validating parameters or preconditions of a function, when you want to indicate that the calling code is wrong somehow:

```
use Carp 'croak';

sub add_two_numbers
{
    croak 'add_two_numbers() takes two and only two arguments'
        unless @_ == 2;
```

```
    ...
}
```

Validating Arguments

Defensive programming often benefits from checking types and values of arguments for appropriateness before performing further processing. By default, Perl 5 provides few built-in mechanisms for doing so (and don't expect *prototypes* to help). You can check that the *number* of parameters passed to a function is correct by evaluating `@_` in scalar context:

```
sub add_numbers
{
    croak "Expected two numbers, but received: " . @_
        unless @_ == 2;

    ...
}
```

Type checking is more difficult, because of Perl's operator-oriented type conversions (see *context_philosophy*). In cases where you need more strictness, consider the CPAN module `Params::Validate`.

Advanced Functions

Functions may seem simple, but you can do much, much more with them (see *closures* and *anonymous_functions* for more details).

Recursion

Every call to a function in Perl creates a new *call frame*. This is an internal data structure which represents the call itself: in effect, incoming parameters, the point to which to return, and all of the control of the program leading up to the point of the call. As well, it captures the lexical environment of the specific and current invocation of the function. This means that a function can *recur*; it can call itself.

Recursion is a deceptively simple concept, but it can seem daunting if you haven't encountered it before. Consider a case where you want to find an element in a sorted array. You *could* iterate through every element of the array individually, looking for the target, but on average, you'll have to look at half of the elements of the array every time.

Another approach is to divide the array in half. Pick the element at the midpoint, compare, then see if you have to divide the lower half or the upper half and continue. You can write this algorithm with a loop yourself, or you could let Perl manage all of the state and tracking necessary with a recursive algorithm. That might look something like:

```
use Modern::Perl;

use Test::More tests => 8;

my @elements = ( 1, 5, 6, 19, 48, 77, 997, 1025, 7777, 8192, 9999 );

ok   elem_exists(     1, @elements ), 'found first element in array';
ok   elem_exists(  9999, @elements ), 'found last element in array';
ok ! elem_exists(   998, @elements ), 'did not find element not in
array';
ok ! elem_exists(    -1, @elements ), 'did not find element not in
array';
ok ! elem_exists( 10000, @elements ), 'did not find element not in
array';
```

```
    ok   elem_exists(    77, @elements ), 'found midpoint element';
    ok   elem_exists(    48, @elements ), 'found end of lower half
element';
    ok   elem_exists(   997, @elements ), 'found start of upper half
element';

    sub elem_exists
    {
        my ($item, @array) = @_;

        # break recursion if there are no elements to search
        return unless @array;

        # bias down, if there are an odd number of elements
        my $midpoint = int( (@array / 2) - 0.5 );
        my $miditem  = $array[ $midpoint ];

        # return true if the current element is the target
        return 1 if $item  == $miditem;

        # return false if the current element is the only element
        return   if @array == 1;

        # split the array down and recurse
        return B<elem_exists>( $item, @array[0 .. $midpoint] )
            if $item < $miditem;

        # split the array up and recurse
        return B<elem_exists>( $item, @array[$midpoint + 1 .. $#array] );
    }
```

This isn't necessarily the best algorithm for searching a sorted list, but it demonstrates recursion. Again, you *can* write this code in a procedural way; but some algorithms are much clearer recursively.

Lexicals

Every new invocation of a function creates its own *instance* of a lexical scope. In the case of the recursive example, even though the declaration of `elem_exists()` creates a single scope for the lexicals `$item`, `@array`, `$midpoint`, and `$miditem`, every *call* to `elem_exists()`, even recursively, has separate storage for the values of those lexical variables. You can demonstrate that by adding debugging code to the function:

```
    B<use Carp 'cluck';>

    sub elem_exists
    {
        my ($item, @array) = @_;

        B<cluck "[$item] (@array)";>

        # other code follows
        ...
    }
```

The output demonstrates that not only can `elem_exists()` call itself safely, but the lexical variables do not interfere with each other.

Tail Calls

One *drawback* of recursion is that you must get your return conditions correct, lest your function call itself an infinite number of times. This is why the `elem_exists()` function has several `return` statements.

Perl offers a helpful warning when it detects what might be runaway recursion: `Deep recursion on subroutine`. The limit is 100 recursive calls, which can be too few in certain circumstances but too many in others. Disable this warning with `no warnings 'recursion'` in the scope of the recursive call.

Because each call to a function requires a new call frame, as well as space for the call to store its own lexical values, highly-recursive code can use more memory than iterative code. A feature called *tail call elimination* can help.

Tail call elimination may be most obvious when writing recursive code, but it can be useful in any case of a tail call. Many programming language implementations support automatic tail call elimination.

A *tail call* is a call to a function which directly returns that function's results. The lines:

```
        # split the array down and recurse
        return elem_exists( $item, @array[0 .. $midpoint] )
            if $item < $miditem;

        # split the array up and recurse
        return elem_exists( $item, @array[$midpoint + 1 .. $#array] );
```

... which return the results of the recursive `elem_exists()` calls directly, are candidates for tail call elimination. This elimination avoids returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Perl 5 supports manual tail call elimination, but two recent CPAN modules provide a much better syntax (and even better performance). Yuval Kogman's `Sub::Call::Tail` is worth exploring if you find yourself with highly recursive code or code that could benefit from tail call elimination. `Sub::Call::Tail` is appropriate for tail calls of non-recursive code:

```
    use Sub::Call::Tail;

    sub log_and_dispatch
    {
        my ($dispatcher, $request) = @_;
        warn "Dispatching with $dispatcher\n";

        return dispatch( $dispatcher, $request );
    }
```

In this example, you can replace the `return` with the new `tail` keyword with no functional changes (yet more clarity and improved performance):

```
        B<tail> dispatch( $dispatcher, $request );
```

If you really *must* perform your own manual tail call elimination, a special form of the `goto` keyword exists. Unlike the form which can often lead to spaghetti code, the `goto` function form replaces the

current function call with a call to another function. You may use a function by name or by reference. You must always set @_ yourself manually, if you want to pass different arguments:

```
# split the array down and recurse
if ($item < $miditem)
{
    @_ = ($item, @array[0 .. $midpoint]);
    B<goto &elem_exists;>
}

# split the array up and recurse
else
{
    @_ = ($item, @array[$midpoint + 1 .. $#array] );
    B<goto &elem_exists;>
}
```

The comparative cleanliness of the CPAN versions is obvious.

Pitfalls and Misfeatures

Not all features of Perl 5 functions are always helpful. In particular, *prototypes* rarely do what you mean. They have their uses, but you can avoid them outside of a few cases.

Check the timeline, but I'm pretty sure about this.

As well, Perl 5 still supports old-style invocations of functions, carried over from older versions of Perl. While you may now invoke Perl functions by name, previous versions of Perl required you to invoke them with a leading ampersand (&) character. Perl 1 required you to use the do keyword as well:

```
# Perl 4 style; avoid
my $result = &calculate_result( 52 );

# Perl 1 style; really truly avoid
my $result = do &calculate_result( 42 );
```

While they're visually noisy, with vestigial syntax, the leading ampersand form performs other behavior which can occasionally surprise you. First, it disables prototype checking (as if that often mattered). Second, if you do not pass arguments explicitly, it *implicitly* passes the contents of @_ unmodified. Both can lead to surprising behavior.

A final pitfall comes from leaving the parentheses off of function calls. The Perl 5 parser uses several heuristics to resolve ambiguity of barewords and the number of parameters passed to a function, but occasionally those heuristics guess wrong. While it's often wise to remove extraneous parentheses, compare the readability of these two lines of code:

```
ok( elem_exists( 1, @elements ), 'found first element in array' );

# warning; contains a subtle bug
ok elem_exists 1, @elements, 'found first element in array';
```

The subtle bug in the second form is that the call to `elem_exists()` will gobble up the test description intended as the second argument to `ok()`. Because `elem_exists()` uses a slurpy second parameter, this may go unnoticed until Perl produces warnings about comparing a non-number (the test description, which it cannot convert into a number) with the element in the array.

This is admittedly an extreme case, but it is a case where proper parenthesization can clarify code and make subtle bugs obvious to the reader.

Add `wantarray()` and Want.pm information?

Scope

Scope in Perl refers to the lifespan and visibility of symbols. Everything with a name in Perl (a variable, a function) has a scope. Scoping helps to enforce *encapsulation*--keeping related concepts together and preventing them from leaking out.

Lexical Scope

The most common form of scoping in modern Perl is lexical scoping. The Perl compiler resolves this scope during compilation. This scope is visible as you *read* a program.

To create a new lexical scope, create a block delimited by curly braces. This block can be a bare block, the block of a loop construct, the block of a `sub` declaration, an `eval` block, or any other non-quoting block:

```
# outer lexical scope
{
    package My::Class;

    # inner lexical scope
    sub awesome_method
    {
        # further inner lexical scope
        do {
            ...
        } while (@_);

        # sibling inner lexical scope
        for (@_)
        {
            ...
        }
    }
}
```

Lexical scope governs the visibility of variables declared with `my`; these are *lexical* variables. A lexical variable declared in one scope is visible in that scope and any scopes nested within it, but is invisible to sibling or outer scopes. Thus, in the code:

```
# outer lexical scope
{
    package My::Class;

    my $outer;

    sub awesome_method
    {
```

```
            my $inner;

            do {
                my $do_scope;
                ...
            } while (@_);

            # sibling inner lexical scope
            for (@_)
            {
                my $for_scope;
                ...
            }
        }
    }
```

... `$outer` is visible in all four scopes. `$inner` is visible in the method, the `do` block, and the `for` loop. `$do_scope` is visible only in the `do` block and `$for_scope` within the `for` loop.

Declaring a lexical in an inner scope with the same name as a lexical in an outer scope hides, or *shadows*, the outer lexical:

```
    {
        my $name = 'Jacob';

        {
            my $name = 'Edward';
            say $name;
        }

        say $name;
    }
```

This program prints `Edward` and then `Jacob`. Even though redeclaring a lexical variable with the same name and type in a single lexical scope produces a warning message, shadowing a lexical in a nested scope does not; this is a feature of lexical shadowing.

Lexical shadowing can happen by accident, but if you limit the scope of variables and limit the nesting of scopes--as is good design anyhow--you lessen your risk.

Lexical declaration has its subtleties. For example, a lexical variable used as the iterator variable of a `for` loop has a scope *within* the loop block. It is not visible outside the block:

```
    my $cat = 'Bradley';

    for my $cat (qw( Jack Daisy Petunia Tuxedo ))
    {
        say "Iterator cat is $cat";
    }

    say "Static cat is $cat";
```

Similarly, the `given` construct creates a *lexical topic* (akin to `my $_`) within its block:

```
$_ = 'outside';

given ('inner')
{
    say;
    $_ = 'whomped inner';
}

say;
```

... despite assignment to $_ inside the block. You may explicitly lexicalize the topic yourself, though this is more useful when considering dynamic scope.

Finally, lexical scoping facilitates closures (see *closures*). Beware creating closures accidentally.

Our Scope

Within a given scope, you may declare an alias to a package variable with the `our` keyword. Like `my`, `our` enforces lexical scoping--of the alias. The fully-qualified name is available everywhere, but the lexical alias is visible only within its scope.

The best use of `our` is for variables you absolutely *must* have, such as `$VERSION`.

Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup happens along the current calling context. Consider the example:

```
{
    our $scope;

    sub inner
    {
        say $scope;
    }

    sub main
    {
        say $scope;
        local $scope = 'main() scope';
        middle();
    }

    sub middle
    {
        say $scope;
        inner();
    }

    $scope = 'outer scope';
    main();
    say $scope;
}
```

The program begins by declaring an `our` variable, `$scope`, as well as three functions. It ends by assigning to `$scope` and calling `main()`.

Within `main()`, the program prints `$scope`'s current value, `outer scope`, then `local`izes the variable. This changes the visibility of the symbol within the current lexical scope *as well as* in any functions called from the current lexical scope. Thus, `$scope` contains `main() scope` within the body of both `middle()` and `inner()`. After `main()` returns--at the point of exiting the block containing the `local`ization of `$scope`, Perl restores the original value of the variable. The final `say` prints `outer scope` once again.

Note that the variable is *visible* within all scopes, but the *value* of the variable changes depending on `local`ization and assignment. This feature can be tricky and subtle, but it is especially useful for changing the values of special variables.

You may only `local`ize global and package global variables. You cannot `local`ize lexical variables.

You *can* localize functions and methods, but that's deep magic to avoid unless absolutely necessary.

Decide whether to use "special" or "magic".

It's common to `local`ize several special variables. For example, `$/`, the input record separator, governs how much data a `readline` operation will read from a filehandle. `$!`, the system error variable, contains the error number of the most recent system call. `$@`, the Perl `eval` error variable, contains any error from the most recent `eval` operation. `$|`, the autoflush variable, governs whether Perl will flush the currently `selected` filehandle after every write operation.

These are all special global variables; `local`izing them in the narrowest possible scope will avoid the action at a distance problem of modifying global variables used other places in your code.

State Scope

A final type of scope is new as of Perl 5.10. This is the scope of the `state` keyword. State scope resembles lexical scope in that it declares a lexical variable, but the value of that variable gets initialized *once*, and then persists:

```
sub counter
{
    B<state> $count = 1;
    return $count++;
}

say counter();
say counter();
say counter();
```

On the first call to state, `$count` has never been initialized, so Perl executes the assignment. The program prints `1`, `2`, and `3`. If you change `state` to `my`, the program will print `1`, `1`, and `1`.

You may also use an incoming parameter to set the initial value of the `state` variable:

```
sub counter
{
    state $count = shift;
    return $count++;
}
```

```
say counter(B<2>);
say counter(B<4>);
say counter(B<6>);
```

Even though a simple reading of the code may suggest that the output should be 2, 4, and 6, the output is actually 2, 3, and 4. The first call to the sub `counter` sets the `$count` variable. Subsequent calls will not change its value. This behavior is as intended and documented, though its implementation can lead to surprising results:

```
sub counter
{
    state $count = shift;
    say 'Second arg is: ', shift;
    return $count++;
}

say counter(2, 'two');
say counter(4, 'four');
say counter(6, 'six');
```

The counter for this program prints 2, 3, and 4 as expected, but the values of the intended second arguments to the `counter()` calls are `two`, 4, and 6--not because the integers are the second arguments passed, but because the `shift` of the first argument only happens in the first call to `counter()`.

`state` can be useful for establishing a default value or preparing a cache, but be sure to understand its initialization behavior if you use it.

Anonymous Functions

An *anonymous function* is a function without a name. It behaves like a named function--you can invoke it, pass arguments to it, return values from it, copy references to it--it can do anything a named function can do. The difference is that it has no name. You always deal with anonymous functions by reference (see *references* and *function_references*).

Declaring Anonymous Functions

You may never declare an anonymous function on its own; you must construct it and assign it to a variable, invoke it immediately, or pass it as an argument to a function, either explicitly or implicitly. Explicit creation uses the `sub` keyword with no name:

```
my $anon_sub = sub { ... };
```

A common Perl 5 idiom known as a *dispatch table* uses hashes to associate input with behavior:

```
my %dispatch =
(
    plus     => sub { $_[0]  + $_[1] },
    minus    => sub { $_[0]  - $_[1] },
    times    => sub { $_[0]  * $_[1] },
    goesinto => sub { $_[0]  / $_[1] },
    raisedto => sub { $_[0] ** $_[1] },
);

sub dispatch
```

```
{
    my ($left, $op, $right) = @_;

    die "Unknown operation!"
        unless exists $dispatch{ $op };

    return $dispatch{ $op }->( $left, $right );
}
```

The `dispatch()` function takes arguments of the form `(2, 'times', 2)` and returns the result of evaluating the operation.

You may use anonymous functions in place of function references. To Perl, they're equivalent. Nothing *necessitates* the use of anonymous functions to perform these mathematical operations, but for functions this short, there's little drawback to writing them this way.

You may rewrite `%dispatch` as:

```
my %dispatch =
(
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    # ... and so on
);

sub add_two_numbers      { $_[0] + $_[1] }

sub subtract_two_numbers { $_[0] - $_[1] }
```

... but the decision to do so depends more on maintainability concerns, safety, and your team's coding style than any language feature.

A benefit of indirection through the dispatch table is that it provides some protection against calling functions without verifying that it's safe to call those functions. If your dispatch function blindly assumed that the string given as the name of the operator corresponded directly to the name of a function to call, a malicious user could conceivably call any function in any other namespace by crafting an operator name of `'Internal::Functions::some_malicious_function'`.

You may also create anonymous functions on the spot when passing them as function parameters:

```
sub invoke_anon_function
{
    my $func = shift;
    return $func->( @_ );
}

sub named_func
{
    say 'I am a named function!';
}

invoke_anon_function( \&named_func );
invoke_anon_function( sub { say 'I am an anonymous function' } );
```

Anonymous Function Names

There is one instance in which you can identify the difference between a reference to a named function and an anonymous function--anonymous functions do not (normally) have names. This may sound subtle and silly and obvious, but introspection shows the difference:

```
package ShowCaller;

use Modern::Perl;

sub show_caller
{
    my ($package, $filename, $line, $sub) = caller(1);
    say "Called from $sub in $package at $filename : $line";
}

sub main
{
    my $anon_sub = sub { show_caller() };
    show_caller();
    $anon_sub->();
}

main();
```

The result may be surprising:

```
Called from ShowCaller::B<main> in ShowCaller at anoncaller.pl : 20
Called from ShowCaller::B<__ANON__> in ShowCaller at anoncaller.pl : 17
```

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. Even though this can be difficult to debug, there are ways around this anonymity.

The CPAN module `Sub::Identify` provides a handful of functions useful to inspect the names of functions, given references to them. `sub_name()` is the most immediately obvious:

```
use Sub::Identify 'sub_name';

sub main
{
    say sub_name( \&main );
    say sub_name( sub {} );
}

main();
```

As you might imagine, the lack of identifying information complicates debugging anonymous functions. The CPAN module `Sub::Name` can help. Its `subname()` function allows you to attach names to anonymous functions:

```
use Sub::Name;
use Sub::Identify 'sub_name';

my $anon  = sub {};
say sub_name( $anon );

my $named = subname( 'pseudo-anonymous', $anon );
```

```
say sub_name( $named );
say sub_name( $anon );

say sub_name( sub {} );
```

This program produces:

```
__ANON__
pseudo-anonymous
pseudo-anonymous
__ANON__
```

Note that both references refer to the same underlying anonymous function. Calling `subname()` on `$anon` and returning into `$named` modifies that function, so any other reference to this function will see the same moniker.

Implicit Anonymous Functions

All of these anonymous function declarations have been explicit. Perl 5 allows implicit anonymous functions through the use of prototypes (*prototypes*). Though this feature exists nominally to enable programmers to write their own syntax such as that for `map` and `eval`, an interesting example is the use of *delayed* functions that don't look like functions. Consider the CPAN module `Test::Exception`:

```
use Test::More tests => 2;
use Test::Exception;

throws_ok { die "I croak!" }
     qr/I croak/, 'die() should throw an exception';

lives_ok  { 1 + 1 }
    'simple addition should not';
```

Both `lives_ok()` and `throws_ok()` take an anonymous function as their first arguments. This code is equivalent to:

```
throws_ok( B<sub { die "I croak!" },>
     qr/I croak/, 'die() should throw an exception' );

lives_ok( B<sub { 1 + 1 },>
    'simple addition should not' );
```

... but is slightly easier to read.

Note the *lack* of a comma following the final curly brace of the implicit anonymous function in the implicit version. This is occasionally a confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl 5 parser.

The implementation of both functions does not care which mechanism you use to pass function references. You can pass named functions by reference as well:

```
B<sub croak { die 'I croak!' }>

B<sub add   { 1 + 1 }>

throws_ok B<\&croak>,
```

```
         qr/I croak/, 'die() should throw an exception';

    lives_ok  B<\&add>,
        'simple addition should not';
```

... but you may *not* pass them as scalar references:

```
    sub croak { die 'I croak!' }

    sub add   { 1 + 1 }

    B<my $croak = \&croak;>
    B<my $add   = \&add;>

    throws_ok B<$croak>,
         qr/I croak/, 'die() should throw an exception';

    lives_ok  B<$add>,
        'simple addition should not';
```

... because the prototype changes the way the Perl 5 parser interprets this code. It cannot determine with 100% clarity *what* `$croak` and `$add` will contain when it evaluates the `throws_ok()` or `lives_ok()` calls, so it produces an error:

```
    Type of arg 1 to Test::Exception::throws_ok must be block or sub {}
    (not private variable) at testex.pl line 13,
    near "'die() should throw an exception';"
```

This feature is occasionally useful despite its drawbacks. The syntactic clarity available by promoting bare blocks to anonymous functions can be helpful, but use it sparingly and document the API with care.

Closures

You've seen how functions work (*functions*). You understand how scope works (*scope*). You know that every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope. You can work with function references (*references*) and anonymous functions (*anonymous_functions*).

You know everything you need to know to understand closures.

Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions, closures, and the amazing things you can do with them. You can read it online at http://hop.perl.plover.com/.

Creating Closures

A *closure* is a function that closes over an outer lexical environment. You've probably already created and used closures without realizing it:

```
    {
        package Invisible::Closure;

        my $filename = shift @ARGV;

        sub get_filename
```

```
        {
            return $filename;
        }
    }
```

The behavior of this code is unsurprising. You may not have noticed anything special. *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works! Yet closures can also close over *transient* lexical environments.

Suppose you want to iterate over a list of items without managing the iterator yourself. You can create a function which returns a function that, when invoked, will return the next item in the iteration:

```
sub make_iterator
{
    my @items = @_;
    my $count = 0;

    return sub
    {
        return if $count == @items;
        return $items[ $count++ ];
    }
}

my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));

say $cousins->() for 1 .. 5;
```

Even though `make_iterator()` has returned, the anonymous function still refers to the lexical variables `@items` and `$count`. Their values persist (*reference_counts*). The anonymous function, stored in `$cousins`, has closed over these values in the specific lexical environment of the specific invocation of `make_iterator()`.

It's easy to demonstrate that the lexical environment is independent between calls to `make_iterator()`:

```
my $cousins = make_iterator(qw( Rick Alex Kaycee Eric Corey ));
my $aunts   = make_iterator(qw( Carole Phyllis Wendy ));

say $cousins->();
say $aunts->();
say $cousins->();
say $aunts->();
```

Because every invocation of `make_iterator()` creates a separate lexical environment for its lexicals, the anonymous sub it creates and returns closes over a unique lexical environment.

Because `make_iterator()` does not return these lexicals by value or by reference, no other Perl code besides the closure can access them. They're encapsulated as effectively as any other lexical encapsulation.

Multiple closures can close over the same lexical variables; this is an idiom used occasionally to provide better encapsulation of what would otherwise be a file global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
```

```
        sub get_private { $private_variable }
    }
```

... but be aware that you cannot *nest* named functions. Named functions have package global scope. Any lexical variables shared between nested functions will go unshared when the outer function destroys its first lexical environmentIf that's confusing to you, imagine the implementation..

The CPAN module `PadWalker` lets you violate lexical encapsulation, but anyone who uses it and breaks your code earns the right to fix any concomitant bugs without your help.

Uses of Closures

Closures can make effective iterators over fixed-size lists, but they demonstrate greater advantages when iterating over a list of items too expensive to refer to directly, either because it represents data which costs a lot to compute all at once or it's too large to fit into memory directly.

Consider a function to create the Fibonacci series as you need its elements. Instead of recalculating the series recursively, use a cache and lazily create the elements you need:

```
sub gen_fib
{
    my @fibs = (0, 1, 1);

    return sub
    {
        my $item = shift;

        if ($item >= @fibs)
        {
            for my $calc ((@fibs - 1) .. $item)
            {
                $fibs[$calc] = $fibs[$calc - 2] + $fibs[$calc - 1];
            }
        }

        return $fibs[$item];
    }
}
```

Every call to the function returned by `gen_fib()` takes one argument, the *n*th element of the Fibonacci series. The function generates all preceding values in the series as necessary, caching them, and returning the requested element. It delays computation until absolutely necessary.

If all you ever need to do is to calculate Fibonacci numbers, this approach may seem overly complex. Consider, however, that the function `gen_fib()` can become amazingly generic: it initializes an array as a cache, performs some custom code to populate arbitrary elements of the cache, and returns the calculated or cached value. If you extract the behavior which calculates Fibonacci values, you can use this code to perform all sorts of cached, lazy iterator behaviors.

In other words, you can extract a function, `generate_caching_closure()`, and rewrite `gen_fib()` in terms of that function:

```
sub gen_caching_closure
{
    my ($calc_element, @cache) = @_;

    return sub
    {
        my $item = shift;
```

```
            $calc_element->($item, \@cache) unless $item < @cache;

            return $cache[$item];
        };
    }

    sub gen_fib
    {
        my @fibs = (0, 1, 1);

        return gen_caching_closure(
            sub
            {
                my ($item, $fibs) = @_;

                for my $calc ((@$fibs - 1) .. $item)
                {
                    $fibs->[$calc] = $fibs->[$calc - 2] + $fibs->[$calc -
1];
                }
            },
            @fibs
        );
    }
```

The program behaves the same way as it did before, but the use of higher order functions and closures allows the separation of the cache initialization behavior from the calculation of the next number in the Fibonacci series in an effective way. Customizing the behavior of code--in this case, `gen_caching_closure()`--by passing in a higher order function allows tremendous flexibility and abstraction.

In one sense, you can consider the builtins `map`, `grep`, and `sort` higher-order functions, especially if you compare them to `gen_caching_closure()`.

Closures and Partial Application

Closures can do more than abstract away structural details. They can allow you to customize specific behaviors. In one sense, they can also *remove* unnecessary genericity. Consider the case of a function which takes several parameters:

```
    sub make_sundae
    {
        my %args = @_;

        my $ice_cream = get_ice_cream( $args{ice_cream} );
        my $banana    = get_banana( $args{banana} );
        my $syrup     = get_syrup( $args{syrup} );
        ...
    }
```

All of the customization possibilities might work very well in your full-sized anchor store in a shopping complex, but if you have a little drive-through ice cream cart near the overpass where you only serve French vanilla ice cream on Cavendish bananas, every time you call `make_sundae()` you have to pass arguments that never change.

A technique called *partial application* binds some arguments to a function such that you can fill in the rest at the point of call. This is easy enough to emulate with closures:

```
my $make_cart_sundae = sub
{
    return make_sundae( @_,
        ice_cream => 'French Vanilla',
        banana    => 'Cavendish',
    );
};
```

Now instead of calling `make_sundae()`, you can invoke `$make_cart_sundae->()` and pass only the interesting arguments, without worrying about forgetting the invariants or passing them incorrectly You can even use `Sub::Install` from the CPAN to install this function into your namespace directly..

State versus Closures

Closures (*closures*) are an easy, effective, and safe way to make data persist between function invocations without using global variables. If you need to share variables between named functions, you can introduce a new scope (*scope*) for only those function declarations:

```
{
    my $safety = 0;

    sub enable_safety  { $safety = 1 }
    sub disable_safety { $safety = 0 }

    sub do_something_awesome
    {
        return if $safety;
        ...
    }
}
```

The encapsulation of functions to toggle the safety allows all three functions to share state without exposing the lexical variable directly to external code. This idiom works well for cases where external code should be able to change internal state, but it's clunkier when only one function needs to manage that state.

Suppose that you want to count the number of customers at your ice cream parlor. Every hundredth person gets free sprinkles:

```
{
    my $cust_count = 0;

    sub serve_customer
    {
        $cust_count++;

        my $order = shift;

        add_sprinkles($order) if $cust_count % 100 == 0)

        ...
    }
```

```
}
```

This approach *works*, but creating a new lexical scope for a single function introduces more accidental complexity than is necessary. The `state` keyword allows you to declare a lexically scoped variable with a value that persists between invocations:

```
sub serve_customer
{
    B<state $cust_count = 0;>
    $cust_count++;

    my $order = shift;
    add_sprinkles($order) if $cust_count % 100 == 0)

    ...
}
```

Note that you must enable this feature explicitly by using a module such as `Modern::Perl`, the `feature` pragma, or requiring a specific version of Perl of 5.10 or newer (with `use 5.010;` or `use 5.012;`, for example).

You may also use `state` within anonymous functions, such as the canonical counter example:

```
sub make_counter
{
    return sub
    {
         B<state $count = 0;>
         return $count++;
     }
}
```

... though there are few obvious benefits to this approach.

Perl 5.10 deprecated a technique from previous versions of Perl by which you could effectively emulate `state`. Using a postfix conditional which evaluates to false with a `my` declaration avoids *reinitializing* a lexical variable to `undef` or its initialized value. In effect, a named function can close over its previous lexical scope by abusing a quirk of implementation.

Any use of a postfix conditional expression modifying a lexical variable declaration now produces a deprecation warning. It's too easy to write inadvertently buggy code with this technique; use `state` instead where available, or a true closure otherwise. Avoid this idiom, but understand it if you encounter it:

```
sub inadvertent_state
{
    # DEPRECATED; do not use
    my $counter = 1 if 0;

    ...
}
```

Attributes

Named entities in Perl--variables and functions--can have additional metadata attached to them in the form of *attributes*. Attributes are names (and, often, values) which allow certain types of metaprogramming (*code_generation*).

Declaring attributes can be awkward, and using them effectively is more art than science. They're relatively rare in most programs for good reason, though they *can* offer compelling benefits of maintenance and clarity.

Using Attributes

In its simplest form, an attribute is a colon-preceded identifier attached to a variable or function declaration:

```
my $fortress      B<:hidden>;

sub erupt_volcano B<:ScienceProject> { ... }
```

These declarations will cause the invocation of attribute handlers named `hidden` and `ScienceProject`, if they exist for the appropriate type (scalars and functions, respectively). If the appropriate handlers do not exist, Perl will throw a compile-time exception. These handlers could do *anything*.

Attributes may include a list of parameters; Perl treats them as a list of constant strings, even if they may resemble other values, such as numbers or variables. The `Test::Class` module from the CPAN uses such parametric arguments to good effect:

```
sub setup_tests :Test(setup) { ... }

sub test_monkey_creation :Test(10) { ... }

sub shutdown_tests :Test(teardown) { ... }
```

The `Test` attribute identifies methods which include test assertions, and optionally identifies the number of assertions the method intends to run. While introspection (*reflection*) of these classes could discover the appropriate test methods, given well-designed solid heuristics, the `:Test` attribute makes the code and its intent unambiguous.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about name clashes or other conflicts due to inheritance or other class design concerns. You *could* enforce a design where all test classes must override methods named `setup()` and `teardown()` themselves, but the attribute approach gives more flexibility of implementation.

The Catalyst web framework also uses attributes to determine the visibility and behavior of methods within web applications.

Drawbacks of Attributes

Attributes do have their drawbacks:

* The canonical pragma for working with attributes (the `attributes` pragma) has listed its interface as experimental for many years. Damian Conway's core module `Attribute::Handlers` simplifies their implementation. Prefer it to `attributes` whenever possible.

* Any module which declares attribute handlers must *inherit* from `Attribute::Handlers` to make the handlers visible to all packages which use themYou *could* also store them in `UNIVERSAL`, but that is global pollution and worse design.. This is due to the implementation of attributes in Perl 5 itself.

* Attribute handlers take effect during `CHECK` blocks, making them inopportune for projects which

themselves manipulate the order of parsing and compilation, such as mod_perl.

* Any arguments provided to attributes are a list of constant strings. `Attribute::Handlers` performs some data conversion, but you may have to disable it occasionally.

The worst feature of attributes is their propensity to produce weird syntactic action at a distance. Given a snippet of code with attributes, can you predict their effect? Good and accurate documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of the destruction of its contents may be wrong, unless you read the documentation very carefully. Likewise, a handler may wrap a function in another function and replace it in the symbol table without your knowledge--consider a `:memoize` attribute which automatically invokes the `Memoize` module.

Complex features can produce compact and idiomatic code. Perl allows developers to experiment with multiple designs to find the best representation for their ideas. Attributes and other advanced Perl features can help you solve complex problems, but they can also obfuscate the intent of code that could otherwise be simple.

Most programs never need this feature.

AUTOLOAD

You do not have to define *every* function and method anyone will ever call. Perl provides a mechanism by which you can intercept calls to functions and methods which do not yet exist. You can use this to define only those functions you need, or to provide interesting error messages and warnings.

Consider the program:

```
#! perl

use Modern::Perl;

bake_pie( filling => 'apple' );
```

When you run it, Perl will throw an exception due to the call to the undefined function `bake_pie()`. Now add a function called `AUTOLOAD()`:

```
sub AUTOLOAD {}
```

Nothing obvious will happen, except that there is no error. The presence of a function named `AUTOLOAD()` in a package tells Perl to call that function whenever normal dispatch for that function or method fails. Change the `AUTOLOAD()` to emit a message to demonstrate this:

```
sub AUTOLOAD { B<say 'In AUTOLOAD()!'> }
```

Basic Features of AUTOLOAD

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` directly. You may manipulate these arguments as you like:

```
sub AUTOLOAD
{
    # pretty-print the arguments
    B<local $" = ', ';>
    B<say "In AUTOLOAD(@_)!">
}
```

The *name* of the undefined function is available in the pseudo-global variable `$AUTOLOAD`:

```
sub AUTOLOAD
{
    B<our $AUTOLOAD;>

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) B<for $AUTOLOAD>!"
}
```

The `our` declaration (*our*) scopes this variable to the body of `AUTOLOAD()`. The variable contains the fully-qualified name of the undefined function. In this case, the function is `main::bake_pie`. A common idiom is to remove the package name:

```
sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    # pretty-print the arguments
    local $" = ', ';
    say "In AUTOLOAD(@_) B<for $name>!"
}
```

Finally, whatever `AUTOLOAD()` returns, the original call receives:

```
say secret_tangent( -1 );

sub AUTOLOAD { return 'mu' }
```

So far, these examples have merely intercepted calls to undefined functions. You have other options.

Redispatching Methods in AUTOLOAD()

A common pattern in OO programming is to *delegate* or *proxy* certain methods in one object to another, often contained in or otherwise accessible from the former. This is an interesting and effective approach to logging:

```
package Proxy::Log;

sub new
{
    my ($class, $proxied) = @_;
    bless \$class, $proxied;
}

sub AUTOLOAD
{
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;
    Log::method_call( $name, @_ );

    my $self = shift;
    return $$self->$name( @_ );
}
```

This `AUTOLOAD()` logs the method call. Its real magic is a simple pattern; it dereferences the proxied

object from a blessed scalar reference, extracts the name of the undefined method, then invokes the method of that name on the proxied object, passing the given arguments.

Generating Code in AUTOLOAD()

That double-dispatch trick is useful, but it is slower than necessary. Every method call on the proxy must go through normal dispatch and fail, then end up in `AUTOLOAD()`. You can instead install new methods into the proxy class as the program needs them:

```
sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    my $method = sub
    {
        Log::method_call( $name, @_ );

        my $self = shift;
        return $self->$name( @_ );
    }

    B<no strict 'refs';>
    B<*{ $AUTOLOAD } = $method;>
    return $method->( @_ );
}
```

The body of the previous `AUTOLOAD()` has become an anonymous function. The code creates a closure (*closures*) bound over the *name* of the undefined method. Then it installs that closure in the appropriate symbol table so that all subsequent dispatch to that method will find the created closure and will avoid `AUTOLOAD()`. Finally, it invokes the method directly and returns the result.

Though this approach is cleaner and almost always more transparent than handling the behavior directly in `AUTOLOAD()`, the code *called* by `AUTOLOAD()` may detect that dispatch has gone through `AUTOLOAD()`. In short, `caller()` will reflect the double-dispatch of both techniques shown so far. This may be an issue; certainly you can argue that it's an encapsulation violation to care, but it's also an encapsulation violation to let the details of *how* an object provides a method to leak out into the wider world.

Another idiom is to use a tailcall (*tail_calls*) to *replace* the current invocation of `AUTOLOAD()` from `caller()`'s memory with a call to the destination method:

```
sub AUTOLOAD
{
    B<my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    my $method = sub { ... }

    no strict 'refs';
    *{ $AUTOLOAD } = $method;
    B<goto &$method;>
}
```

This has the same effect as invoking `$method` directly, except that `AUTOLOAD()` will no longer appear in the list of calls available from `caller()`, so it looks like the generated method was simply called directly.

Drawbacks of AUTOLOAD

`AUTOLOAD()` can be a useful tool in certain circumstances, but it can be difficult to use properly. Consider other techniques, such as `Moose` and other abstractions, instead.

The naïve approach to generating methods at runtime means that the `can()` method will not report the right information about the capabilities of objects and classes. You can solve this in several ways; one of the easiest is to pre-declare all functions you plan to AUTOLOAD with the `subs` pragma:

```
use subs qw( red green blue ochre teal );
```

That technique has the advantage of documenting your intent but the disadvantage that you have to maintain a static list of functions or methods.

You can also provide your own `can()` to generate the appropriate functions:

```
sub can
{
    my ($self, $method) = @_;

    # use results of parent can()
    my $meth_ref = $self->SUPER::can( $method );
    return $meth_ref if $meth_ref;

    # add some filter here
    return unless $self->should_generate( $method );

    $meth_ref = sub { ... };
    no strict 'refs';
    return *{ $method } = $meth_ref;
}

sub AUTOLOAD
{
    my ($self) = @_;
    my ($name) = our $AUTOLOAD =~ /::(\w+)$/;>

    return unless my $meth_ref = $self->can( $name );
    goto &$meth_ref;
}
```

Depending on the complexity of your needs, you may find it easier to maintain a data structure such as a package-scoped hash which contains acceptable names of methods to generate.

Be aware that certain methods you do not intend to provide may go through `AUTOLOAD()`. A common culprit is `DESTROY()`, the destructor of objects. The simplest approach is to provide a `DESTROY()` method with no implementation; Perl will happily dispatch to this and ignore `AUTOLOAD()` altogether:

```
# skip AUTOLOAD()
sub DESTROY {}
```

Special methods such as `import()`, `unimport()`, and `VERSION()` never go through `AUTOLOAD()`.

If you mix functions and methods in a single namespace which inherits from another package which provides its own `AUTOLOAD()`, you may get a strange error message:

```
Use of inherited AUTOLOAD for non-method I<slam_door>() is deprecated
```

This occurs when you try to call a function which does not exist in a package which inherits from a class which contains its own `AUTOLOAD()`. This is almost never what you intend. The problem compounds in several ways: mixing functions and methods in a single namespace is often a design flaw, inheritance and `AUTOLOAD()` get complex very quickly, and reasoning about code when you don't know what methods objects can perform is difficult.