A *function* (or *subroutine*) in Perl is a discrete, encapsulated unit of behavior. A program is a collection of little black boxes where the interaction of these functions governs the control flow of the program. A function may have a name. It may consume incoming information. It may produce outgoing information.

Functions are a prime mechanism for abstraction, encapsulation, and re-use in Perl 5.

Declaring Functions

Use the `sub` builtin to declare a function:

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

Now `greet_me()` is available for invocation anywhere else within the program.

You do not have to *define* a function at the point you declare it. A *forward declaration* tells Perl to remember the function name even though you will define it later:

```
sub greet_sun;
```

Invoking Functions

Use postfix (*fixity*) parentheses and a function's name to invoke that function and pass an optional list of arguments:

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

While these parentheses are not strictly necessary for these examples--even with `strict` enabled--they provide clarity to human readers and Perl's parser. When in doubt, leave them in.

Function arguments can be arbitrary expressions, including simple variables:

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

... though Perl 5's default parameter handling sometimes surprises novices.

Function Parameters

A function receives its parameters in a single array, `@_` (*default_array_variables*). Perl *flattens* all incoming parameters into a single list. The function must either unpack all parameters into variables or operate on `@_` directly:

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

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

Most code uses `shift` or list unpacking, though `@_` behaves as a normal Perl array, so you can refer to individual elements by index:

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

sub greet_two_no_shift
{
    my ($hero, $sidekick) = @_;
    say "Well if it isn't $hero and $sidekick. Welcome!";
}

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

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

... or `shift`, `unshift`, `push`, `pop`, `splice`, and slice `@_`.

Remember that the array builtins use `@_` as the default operand *within functions*. Take advantage of this idiom.

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 look 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 = B<shift>;
    say 'Calling delegated_method()'

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

Use `shift` when your function needs only a single parameter. Use list assignment when accessing multiple parameters.

Several CPAN distributions extend Perl 5's parameter handling with additional syntax and options. `signatures` and `Method::Signatures` are powerful. `Method::Signatures::Simple` is basic, but useful. `MooseX::Method::Signatures` works very well with Moose (*moose*).

Flattening

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

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

show_pets( %pet_names_and_types );

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

When Perl flattens `%pet_names_and_types` into a list, the order of the key/value pairs from the hash will vary, but the list will always contain a key immediately followed by its value. Hash assignment inside `show_pets()` works essentially as the more explicit assignment to `%pet_names_and_types` does.

This flattening is often useful, but beware of mixing scalars with flattened aggregates in parameter lists. To write a `show_pets_of_type()` function, where one parameter is the type of pet to display, 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

List assignment with an aggregate is always greedy, so assigning to `%pets` *slurps* all of the remaining values from `@_`. If the `$type` parameter came at the end of `@_`, Perl would warn about assigning an odd number of elements to the hash. You *could* work around that:

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

    ...
}
```

... at the expense of clarity. The same principle applies when assigning to an array as a parameter, of course. Use references (*references*) to avoid unwanted aggregate flattening and slurping.

Aliasing

`@_` contains a subtlety; it *aliases* function arguments such that you can modify them directly. For example:

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

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

# prints C<egnarO>
```

Modify an element of `@_` directly and you will modify the original parameter. Be cautious, and unpack `@_` rigorously.

Functions and Namespaces

Every function has a containing namespace (*packages*). Functions in an undeclared namespace--functions not declared after an explicit `package` statement--are in the `main` namespace. You may also declare a function within another namespace by prefixing its name:

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

This will declare the function and create the namespace as necessary. Remember that Perl 5 packages are open for modification at any point. You may only declare one function of the same name per namespace. Otherwise Perl 5 will warn you about subroutine redefinition. Disable this warning with `no warnings 'redefine'`--if you're certain this is what you intend.

Call functions in other namespaces with their fully-qualified names:

```
package main;

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

Functions in namespaces are *visible* outside of those namespaces through their fully-qualified names. Within a namespace, you may use the short name to call any function declared in that namespace. You may also import names from other namespaces.

Importing

When loading a module with the `use` builtin (*modules*), Perl automatically calls a method named `import()` on that module. Modules can provide their own `import()` which makes some or all defined symbols available to the calling package. 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' )`.

`use` has special behavior with regard to `import()`, but you may call `import()` directly. The `use` example is equivalent to:

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

The `use` builtin adds an implicit `BEGIN` block around these statements so that the `import()` call happens *immediately* after the parser has compiled the entire `use` 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 barewords, and would violate `strict`.

Reporting Errors

Within a function, inspect the context of the call to the function with the `caller` builtin. When 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 file 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 $file:$line";
}
```

The full call chain is available for inspection. Pass a single integer argument *n* to `caller()` to inspect the caller of the caller of the caller *n* times. 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.

This optional argument also tells `caller` to provide additional 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 $file:$line";
}
```

The standard `Carp` module uses this technique to great effect for reporting errors and throwing warnings in functions. When used in place of `die` in library code, `croak()` throws an exception from the point of view of its caller. `carp()` reports a warning from the file and line number of its caller (*producing_warnings*).

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

Validating Arguments

While Perl does its best to do what the programmer means, it offers few native ways to test the validity of arguments provided to a function. Evaluate `@_` in scalar context to check that the *number* of parameters passed to a function is correct:

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

    ...
}
```

Type checking is more difficult, because of Perl's operator-oriented type conversions (*context_philosophy*). The CPAN module `Params::Validate` offers more strictness.

Advanced Functions

Functions are the foundation of many advanced Perl features.

Context Awareness

Perl 5's builtins know whether you've invoked them in void, scalar, or list context. So too can your functions. The misnamedSee `perldoc -f wantarray` to verify. `wantarray` builtin returns `undef` to signify void context, a false value to signify scalar context, and a true value to signify list context.

```
sub context_sensitive
{
    my $context = wantarray();

    return qw( List context )   if         $context;
    say    'Void context'   unless defined $context;
    return 'Scalar context' unless         $context;
}

context_sensitive();
say my $scalar = context_sensitive();
say context_sensitive();
```

This can be useful for functions which might produce expensive return values to avoid doing so in void context. Some idiomatic functions return a list in list context and the first element of the list or an array reference in scalar context. Yet remember that there exists no single best recommendation for the use `wantarray`. Sometimes it's clearer to write separate and unambiguous functions.

Robin Houston's `Want` and Damian Conway's `Contextual::Return` distributions from the CPAN offer many possibilities for writing powerful and usable context-aware interfaces.

Recursion

Suppose 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 examine half of the elements of the array. Another approach is to halve the array, pick the element at the midpoint, compare, then repeat with either the lower or upper half. Divide and conquer. When you run out of elements to inspect or find the element, stop.

An automated test for this technique could be:

```
use Test::More;

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';

done_testing();
```

Recursion is a deceptively simple concept. Every call to a function in Perl creates a new *call frame*, an internal data structure which represents the call itself, including the lexical environment of the function's current invocation. This means that a function can call itself, or *recur*.

To make the previous test pass, write a function called `elem_exists()` which knows how to call itself, halving the list each time:

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

    # break recursion with no elements to search
    return unless @array;

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

    # return true if found
    return 1 if $item  == $miditem;

    # return false with only one 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 and recurse
    return B<elem_exists>(
         $item, @array[ $midpoint + 1 .. $#array ]
    );
}
```

While you *can* write this code in a procedural way and manage the halves of the list yourself, this recursive approach lets Perl manage the bookkeeping.

Lexicals

Every new invocation of a function creates its own *instance* of a lexical scope. 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--stores the values of those lexicals separately.

Not only can `elem_exists()` call itself, but the lexical variables of each invocation are safe and separate:

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

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

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

    # other code follows
    ...
}
```

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. `elem_exists()` function has several `return` statements for this reason.

Perl offers a helpful `Deep recursion on subroutine` warning when it suspects runaway recursion. The limit of 100 recursive calls is arbitrary, but often useful. 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 and lexical storage space, highly-recursive code can use more memory than iterative code. *Tail call elimination* can help.

A *tail call* is a call to a function which directly returns that function's results. These recursive calls to `elem_exists()`:

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


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

... are candidates for tail call elimination. This optimization would avoid returning to the current call and then returning to the parent call. Instead, it returns to the parent call directly.

Unfortunately, Perl 5 does not eliminate tail calls automatically. Do so manually with a special form of the `goto` builtin. Unlike the form which often produces 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. To pass different arguments, assign to `@_` directly:

```
# 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;>
    }
```

Sometimes optimizations are ugly.

Pitfalls and Misfeatures

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` builtin:

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

# Perl 1 style; avoid
my $result = do calculate_result( 42 );

# crazy mishmash; really truly avoid
my $result = do &calculate_result( 42 );
```

While the vestigial syntax is visual clutter, the leading ampersand form has other surprising behaviors. First, it disables any prototype checking. Second, it *implicitly* passes the contents of `@_` unmodified unless you specify arguments yourself. 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 ambiguous barewords and the number of parameters passed to a function. Heuristics can be wrong:

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

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.

While extraneous parentheses can hamper readability, thoughtful use of parentheses can clarify code and make subtle bugs unlikely.

Scope

Scope in Perl refers to the lifespan and visibility of named entities. 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

Lexical scope is the scope visible as you *read* a program. The Perl compiler resolves this scope during compilation. A block delimited by curly braces creates a new scope, whether a bare block, the block of a loop construct, the block of a `sub` declaration, an `eval` block, or any other non-quoting block.

Lexical scope governs the visibility of variables declared with `my`--*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:

```
# outer lexical scope
{
    package Robot::Butler

    # inner lexical scope
    my $battery_level;

    sub tidy_room
    {
        # further inner lexical scope
        my $timer;

        do {
            # innermost lexical scope
            my $dustpan;
            ...
        } while (@_);

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

... `$battery_level` is visible in all four scopes. `$timer` is visible in the method, the `do` block, and the `for` loop. `$dustpan` is visible only in the `do` block and `$polish_cloth` 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 within the inner scope. This is often what you want:

```
my $name = 'Jacob';

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

say $name;
```

Lexical shadowing can happen by accident. Limit the scope of variables and the nesting of scopes to lessen your risk.

This program prints `Edward` and then `Jacob`Family members, not vampires., even though redeclaring a lexical variable with the same name and type *in the same lexical scope* produces a warning message. Shadowing a lexical is a feature of encapsulation.

Some lexical declarations have subtleties, such as a lexical variable used as the iterator variable of a `for` loop. Its declaration comes outside of the block, but its scope is that *within* the loop block:

```
my $cat = 'Brad';

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

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

Similarly, `given` (*given_when*) creates a *lexical topic* (like `my $_`) within its block:

```
$_ = 'outside';

given ('inner')
{
    say;
    $_ = 'this assignment does nothing useful';
}

say;
```

... such that leaving the block restores the previous value of `$_`.

Functions--named and anonymous--provide lexical scoping to their bodies. This facilitates closures (*closures*).

Our Scope

Within given scope, declare an alias to a package variable with the `our` builtin. 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.

`our` is most useful with package global variables like `$VERSION` and `$AUTOLOAD`.

Dynamic Scope

Dynamic scope resembles lexical scope in its visibility rules, but instead of looking outward in compile-time scopes, lookup traverses backwards through the calling context. While a package global variable may be *visible* within all scopes, its *value* changes depending on `local`ization and assignment:

```
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, when control flow reaches the end of its block, Perl restores the original value of the `local`ized `$scope`. The final `say` prints `outer scope` once again.

Package variables and lexical variables have different visibility rules and storage mechanisms within Perl. Every scope which contains lexical variables has a special data structure called a *lexical pad* or *lexpad* which can store the values for its enclosed lexical variables. Every time control flow enters one of these scopes, Perl creates another lexpad for the values of those lexical variables for that particular call. This makes functions work correctly, especially in recursive calls (*recursion*).

Each package has a single *symbol table* which holds package variables as well as named functions. Importing (*importing*) works by inspecting and manipulating this symbol table. So does `local`. You may only `local`ize global and package global variables--never lexical variables.

`local` is most often useful with magic 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 `select`ed filehandle after every write operation.

`local`izing these in the narrowest possible scope limits the effect of your changes. This can prevent strange behavior in other parts of your code.

State Scope

Perl 5.10 added a new scope to support the `state` builtin. State scope resembles lexical scope in terms of visibility, but adds a one-time initialization as well as value persistence:

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

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

On the first call to `counter`, Perl performs its single initialization of `$count`. On subsequent calls, `$count` retains its previous value. This program prints `1`, `2`, and `3`. Change `state` to `my` and the program will print `1`, `1`, and `1`.

You may use an expression to set a `state` variable's initial value:

```
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.

`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:

```
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`--because the `shift` of the first argument only happens in the first call to `counter()`. Either change the API to prevent this mistake, or guard against it with:

```
sub counter
{
    my ($initial_value, $text) = @_;

    state $count = $initial_value;
    say "Second arg is: $text";
```

```
    return $count++;
}

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

Anonymous Functions

An *anonymous function* is a function without a name. It otherwise behaves exactly like a named function--you can invoke it, pass arguments to it, return values from it, and copy references to it. Yet the only way to deal with it is by reference (*function_references*).

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

```
my %dispatch =
(
    plus     => \&add_two_numbers,
    minus    => \&subtract_two_numbers,
    times    => \&multiply_two_numbers,
);

sub add_two_numbers      { $_[0] + $_[1] }
sub subtract_two_numbers { $_[0] - $_[1] }
sub multiply_two_numbers { $_[0] * $_[1] }

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

    return 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.

Declaring Anonymous Functions

The `sub` builtin used without a name creates and returns an anonymous function. Use this function reference any place you'd use a reference to a named function, such as to declare the dispatch table's functions in place:

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

This dispatch table offers some degree of security; only those functions mapped within the table are available for users to call. 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 passing `'Internal::Functions::malicious_function'`.

You may also see anonymous functions passed as function arguments:

```
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( B<sub { say 'Who am I?' }> );
```

Anonymous Function Names

Given a reference to a function, you can determine whether the function is named or anonymous with introspection... or `sub_name` from the CPAN module `Sub::Identify`.:

```
package ShowCaller;

sub show_caller
{
    my ($package, $file, $line, $sub) = caller(1);
    say "Called from $sub in $package:$file:$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:anoncaller.pl:20
Called from ShowCaller::B<__ANON__>
         in ShowCaller:anoncaller.pl:17
```

The `__ANON__` in the second line of output demonstrates that the anonymous function has no name that Perl can identify. This can complicate debugging. The CPAN module `Sub::Name`'s `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__
```

Be aware that both references refer to the same underlying anonymous function. Using `subname()` on one reference to a function will modify that anonymous function's name such that all other references to it will see the new name.

Implicit Anonymous Functions

Perl 5 allows the declaration of anonymous functions implicitly 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::Fatal`, which takes an anonymous function as the first argument to its `exception()` function:

```
use Test::More;
use Test::Fatal;

my $croaker = exception { die 'I croak!' };
my $liver   = exception { 1 + 1 };

like( $croaker, qr/I croak/, 'die() should croak'   );
is(   $liver,   undef,       'addition should live' );

done_testing();
```

You might rewrite this more verbosely as:

```
my $croaker = exception( sub { die 'I croak!' } );
my $liver   = exception( sub { 1 + 1 } );
```

... or to pass named functions by reference:

```
B<sub croaker { die 'I croak!' }>
B<sub liver   { 1 + 1 }>
```

```
my $croaker = exception \&croaker;
my $liver   = exception \&liver;

like( $croaker, qr/I croak/, 'die() should die'     );
is(   $liver,   undef,       'addition should live' );
```

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

```
B<my $croak_ref = \&croaker;>
B<my $live_ref  = \&liver;>

# BUGGY: does not work
my $croaker   = exception $croak_ref;
my $liver     = exception $live_ref;
```

... because the prototype changes the way the Perl 5 parser interprets this code. It cannot determine with 100% clarity *what* `$croaker` and `$liver` will contain, and so will throw an exception.

```
Type of arg 1 to Test::Fatal::exception
   must be block or sub {} (not private variable)
```

Also be aware that a function which takes an anonymous function as the first of multiple arguments cannot have a trailing comma after the function block:

```
use Test::More;
use Test::Fatal 'dies_ok';

dies_ok { die 'This is my boomstick!' }
        'No movie references here';
```

This is an occasionally confusing wart on otherwise helpful syntax, courtesy of a quirk of the Perl 5 parser. 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

Every time control flow enters a function, that function gets a new environment representing that invocation's lexical scope (*scope*). That applies equally well to anonymous functions (*anonymous_functions*). The implication is powerful. The computer science term *higher order functions* refers to functions which manipulate other functions. Closures show off this power.

Creating Closures

A *closure* is a function that uses lexical variables from an outer scope. You've probably already created and used closures without realizing it:

```
package Invisible::Closure;

my $filename = shift @ARGV;

sub get_filename { return $filename }
```

If this code seems straightforward to you, good! *Of course* the `get_filename()` function can see the `$filename` lexical. That's how scope works!

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 Mandy Christine )
);

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

Even though `make_iterator()` has returned, the anonymous function, stored in `$cousins`, has closed over the values of these variables as they existed within the invocation of `make_iterator()`. Their values persist (*reference_counts*).

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

```
my $aunts = make_iterator(
    qw( Carole Phyllis Wendy Sylvia Monica Lupe )
);

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

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, although any code which shares a lexical environment can access these values. This idiom provides better encapsulation of what would otherwise be a file or package global variable:

```
{
    my $private_variable;

    sub set_private { $private_variable = shift }
    sub get_private { $private_variable }
}
```

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

Iterating over a fixed-sized list with a closure is interesting, but closures can do much more, such as iterating over a list which is too expensive to calculate or too large to maintain in memory all at once. 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);

    return sub
    {
        my $item = shift;

        if ($item >= @fibs)
        {
            for my $calc (@fibs .. $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, caches them, and returns the requested element--even delaying computation until absolutely necessary. Yet there's a pattern specific to caching intertwined with the numeric series. What happens if you extract the cache-specific code (initialize a cache, execute custom code to populate cache elements, and return the calculated or cached value) to a function `generate_caching_closure()`?

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

    return sub
    {
        my $item = shift;

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

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

The builtins `map`, `grep`, and `sort` are themselves higher-order functions.

Now `gen_fib()` can become:

```
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 as it did before, but the use of function references and closures separates the cache initialization behavior from the calculation of the next number in the Fibonacci series. Customizing the behavior of code--in this case, `gen_caching_closure()`--by passing in a function allows tremendous flexibility.

Closures and Partial Application

Closures can also *remove* unwanted 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} );
    ...
}
```

Myriad customization possibilities might work very well in a full-sized ice cream store, but for a drive-through ice cream cart where you only serve French vanilla ice cream on Cavendish bananas, every call to `make_sundae()` passes arguments that never change.

A technique called *partial application* allows you to bind *some* of the arguments to a function so that you can provide the others later. Wrap the function you intend to call in a closure and pass the bound arguments.

Consider an ice cream cart which only serves French Vanilla ice cream over Cavendish bananas:

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

Instead of calling `make_sundae()` directly, invoke the function reference in `$make_cart_sundae` and pass only the interesting arguments, without worrying about forgetting the invariants or passing

them incorrectlyYou can even use `Sub::Install` from the CPAN to import this function into another namespace directly..

This is only the start of what you can do with higher order functions. Mark Jason Dominus's *Higher Order Perl* is the canonical reference on first-class functions and closures in Perl. Read it online at http://hop.perl.plover.com/.

State versus Closures

Closures (*closures*) take advantage of lexical scope (*scope*) to mediate access to semi-private variables. Even named functions can take advantage of lexical bindings:

```
{
    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 every hundredth person at your ice cream parlor 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` builtin 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);

    ...
}
```

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

`state` also works within anonymous functions:

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

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

State versus Pseudo-State

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

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. Rewrite this idiom when you encounter it:

```
sub inadvertent_state
{
    # my $counter  = 1 if 0; # DEPRECATED; don't use
    state $counter = 1;      # use instead

    ...
}
```

Attributes

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

Attribute declaration syntax is awkward, and using attributes effectively is more art than science. Most programs never use them, but when used well they offer clarity and maintenance benefits.

Using Attributes

A simple attribute is a colon-preceded identifier attached to a 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). These handlers can do *anything*. If the appropriate handlers do not exist, Perl will throw a compile-time exception.

Attributes may include a list of parameters. Perl treats these parameters as lists of constant strings and only strings. 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 your intent clear.

The `setup` and `teardown` parameters allow test classes to define their own support methods without worrying about conflicts with other such methods in other classes. This separates the concern of specifying what this class must do with the concern of how other classes do their work, and offers great flexibility.

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

Drawbacks of Attributes

Attributes 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. Andrew Main's `Attribute::Lexical` is a newer approach. Prefer either to `attributes` whenever possible.

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? Well written documentation helps, but if an innocent-looking declaration on a lexical variable stores a reference to that variable somewhere, your expectations of its lifespan may be wrong. 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 core `Memoize` module.

Attributes are available when you need them to solve difficult problems. They can be very useful, used properly--but most programs never need them.

AUTOLOAD

Perl does not require you to declare every function before you call it. Perl will happily attempt to call a function even if it doesn't exist. Consider the program:

```
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 {}
```

When you run the program now, nothing obvious will happen. Perl will call a function named `AUTOLOAD()` in a package--if it exists--whenever normal dispatch fails. Change the `AUTOLOAD()` to emit a message:

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

... to demonstrate that it gets called.

Basic Features of AUTOLOAD

The `AUTOLOAD()` function receives the arguments passed to the undefined function in `@_` directly and the *name* of the undefined function is available in the package global `$AUTOLOAD`. Manipulate these arguments as you like:

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

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

The `our` declaration (*our*) scopes `$AUTOLOAD` to the function body. The variable contains the fully-qualified name of the undefined function (in this case, `main::bake_pie`). Remove the package name with a regular expression (*regex*):

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

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

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 (*moose*) is to *delegate* or *proxy* certain methods in one object to another, often contained in or otherwise accessible from the former. A logging proxy can help with debugging:

```
package Proxy::Log;
```

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

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

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

This `AUTOLOAD()` logs the method call. Then it dereferences the proxied object from a blessed scalar reference, extracts the name of the undefined method, then invokes that method on the proxied object with the provided parameters.

Generating Code in AUTOLOAD()

This double dispatch is easy to write but inefficient. Every method call on the proxy must fail normal dispatch to end up in `AUTOLOAD()`. Pay that penalty only once by installing 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 a closure (*closures*) bound over the *name* of the undefined method. Installing that closure in the appropriate symbol table allows all subsequent dispatch to that method to find the created closure (and avoid `AUTOLOAD()`). This code finally 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. While it may violate encapsulation to care that this occurs, leaking the details of *how* an object provides a method may also violate encapsulation.

Some code uses a tailcall (*tailcalls*) to *replace* the current invocation of `AUTOLOAD()` 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 useful tool, though it is difficult to use properly. 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. The easiest solution is to predeclare all functions you plan to `AUTOLOAD()` with the `subs` pragma:

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

Forward declarations are only useful in the two rare cases of attributes and autoloading (*autoload*).

That technique has the advantage of documenting your intent but the disadvantage that you have to maintain a static list of functions or methods. Overriding `can()` (*universal*) sometimes works better:

```
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;
}
```

`AUTOLOAD()` is a big hammer; it can catch functions and methods you had no intention of

autoloading, such as DESTROY(), the destructor of objects. If you write a DESTROY() method with no implementation, Perl will happily dispatch to it instead of AUTOLOAD():

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

The special methods 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 see the strange error:

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

If this happens to you, simplify your code; you've called a function which does not exist in a package which inherits from a class which contains its own AUTOLOAD(). 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 provide is difficult.

AUTOLOAD() is useful for quick and dirty programming, but robust code avoids it.