Perl's text processing power comes from its use of *regular expressions*. A regular expression (*regex* or *regexp*) is a *pattern* which describes characteristics of a piece of text. A *regular expression engine* interprets patterns and applies them to match or modify pieces of text.

Perl's core regex documentation includes a tutorial (`perldoc perlretut`), a reference guide (`perldoc perlreref`), and full documentation (`perldoc perlre`). Jeffrey Friedl's book *Mastering Regular Expressions* explains the theory and the mechanics of how regular expressions work. While mastering regular expressions is a daunting pursuit, a little knowledge will give you great power.

Literals

Regexes can be as simple as substring patterns:

```
my $name = 'Chatfield';
say 'Found a hat!' if $name =~ B</hat/>;
```

The match operator (`m//`, abbreviated `//`) identifies a regular expression--in this example, `hat`. This pattern is *not* a word. Instead it means "the h character, followed by the a character, followed by the `t` character." Each character in the pattern is an indivisible element, or *atom*. It matches or it doesn't.

The regex binding operator (=~) is an infix operator (*fixity*) which applies the regex of its second operand to a string provided by its first operand. When evaluated in scalar context, a match evaluates to a true value if it succeeds. The negated form of the binding operator (`!~`) evaluates to a true value unless the match succeeds.

The `index` builtin can also search for a literal substring within a string. Using a regex engine for that is like flying your autonomous combat helicopter to the corner store to buy cheese--but Perl allows you to decide what you find most maintainable.

The substitution operator, `s///`, is in one sense a circumfix operator (*fixity*) with two operands. Its first operand is a regular expression to match when used with the regex binding operator. The second operand is a substring used to replace the matched portion of the first operand used with the regex binding operator. For example, to cure pesky summer allergies:

```
my $status = 'I feel ill.';
$status    =~ s/ill/well/;
say $status;
```

The qr// Operator and Regex Combinations

The `qr//` operator creates first-class regexes. Interpolate them into the match operator to use them:

```
my $hat = B<qr/hat/>;
say 'Found a hat!' if $name =~ /$hat/;
```

... or combine multiple regex objects into complex patterns:

```
my $hat   = qr/hat/;
```

```
my $field = qr/field/;

say 'Found a hat in a field!'
    if $name =~ /B<$hat$field>/;

like( $name, qr/B<$hat$field>/,
               'Found a hat in a field!' );
```

`Test::More`'s `like` function tests that the first argument matches the regex provided as the second argument.

Quantifiers

Regular expressions get more powerful through the use of *regex quantifiers*, which allow you to specify how often a regex component may appear in a matching string. The simplest quantifier is the *zero or one quantifier*, or `?`:

```
my $cat_or_ct = qr/caB<?>t/;

like( 'cat', $cat_or_ct, "'cat' matches /ca?t/" );
like( 'ct',  $cat_or_ct, "'ct' matches /ca?t/"  );
```

Any atom in a regular expression followed by the ? character means "match zero or one of this atom." This regular expression matches if zero or one `a` characters immediately follow a `c` character and immediately precede a `t` character, either the literal substring `cat` or `ct`.

The *one or more quantifier*, or `+`, matches only if there is at least one of the quantified atom:

```
my $some_a = qr/caB<+>t/;

like( 'cat',    $some_a, "'cat' matches /ca+t/" );
like( 'caat',   $some_a, "'caat' matches/"      );
like( 'caaat',  $some_a, "'caaat' matches"      );
like( 'caaaat', $some_a, "'caaaat' matches"     );

unlike( 'ct',   $some_a, "'ct' does not match"  );
```

There is no theoretical limit to the maximum number of quantified atoms which can match.

The *zero or more quantifier*, `*`, matches zero or more instances of the quantified atom:

```
my $any_a = qr/caB<*>t/;

like( 'cat',    $any_a, "'cat' matches /ca*t/" );
like( 'caat',   $any_a, "'caat' matches"       );
like( 'caaat',  $any_a, "'caaat' matches"      );
like( 'caaaat', $any_a, "'caaaat' matches"     );
like( 'ct',     $any_a, "'ct' matches"         );
```

As silly as this seems, it allows you to specify optional components of a regex. Use it sparingly, though: it's a blunt and expensive tool. *Most* regular expressions benefit from using the ? and +

quantifiers far more than *. Precision of intent often improves clarity.

Numeric quantifiers express specific numbers of times an atom may match. `{n}` means that a match must occur exactly *n* times.

```
# equivalent to qr/cat/;
my $only_one_a = qr/caB<{1}>t/;

like( 'cat', $only_one_a, "'cat' matches /ca{1}t/" );
```

`{n,}` matches an atom *at least n* times:

```
# equivalent to qr/ca+t/;
my $some_a = qr/caB<{1,}>t/;

like( 'cat',    $some_a, "'cat' matches /ca{1,}t/" );
like( 'caat',   $some_a, "'caat' matches"          );
like( 'caaat',  $some_a, "'caaat' matches"         );
like( 'caaaat', $some_a, "'caaaat' matches"        );
```

`{n,m}` means that a match must occur at least *n* times and cannot occur more than *m* times:

```
my $few_a = qr/caB<{1,3}>t/;

like( 'cat',    $few_a, "'cat' matches /ca{1,3}t/" );
like( 'caat',   $few_a, "'caat' matches"           );
like( 'caaat',  $few_a, "'caaat' matches"          );

unlike( 'caaaat', $few_a, "'caaaat' doesn't match" );
```

You may express the symbolic quantifiers in terms of the numeric quantifiers, but most programs use the former far more often than the latter.

Greediness

The + and * quantifiers are *greedy*, as they try to match as much of the input string as possible. This is particularly pernicious. Consider a naïve use of the "zero or more non-newline characters" pattern of `.*`:

```
# a poor regex
my $hot_meal = qr/hot.*meal/;

say 'Found a hot meal!'
    if 'I have a hot meal' =~ $hot_meal;

say 'Found a hot meal!'
     if 'one-shot, piecemeal work!' =~ $hot_meal;
```

Greedy quantifiers start by matching *everything* at first, and back off a character at a time only when it's obvious that the match will not succeed.

The ? quantifier modifier turns a greedy-quantifier parsimonious:

```
my $minimal_greedy = qr/hot.*?meal/;
```

When given a non-greedy quantifier, the regular expression engine will prefer the *shortest* possible potential match and will increase the number of characters identified by the `.*?` token combination only if the current number fails to match. Because `*` matches zero or more times, the minimal potential match for this token combination is zero characters:

```
say 'Found a hot meal'
if 'ilikeahotmeal' =~ /$minimal_greedy/;
```

Use +? to match one or more items non-greedily:

```
my $minimal_greedy_plus = qr/hot.+?meal/;

unlike( 'ilikeahotmeal', $minimal_greedy_plus );

like( 'i like a hot meal', $minimal_greedy_plus );
```

The ? quantifier modifier also applies to the ? (zero or one matches) quantifier as well as the range quantifiers. In every case, it causes the regex to match as little of the input as possible.

The greedy patterns `.+` and `.*` are tempting but dangerous. A cruciverbalistA crossword puzzle afficionado. who needs to fill in four boxes of 7 Down ("Rich soil") will find too many invalid candidates with the pattern:

```
my $seven_down   = qr/l$letters_only*m/;
```

She'll have to discard `Alabama`, `Belgium`, and `Bethlehem` long before the program suggests `loam`. Not only are those words too long, but the matches start in the middle of the words. A working understanding of greediness helps, but there is no substitute for the copious testing with real, working data.

Regex Anchors

Regex anchors force the regex engine to start or end a match at an absolute position. The *start of string anchor* (`\A`) dictates that any match must start at the beginning of the string:

```
# also matches "lammed", "lawmaker", and "layman"
my $seven_down = qr/\Al${letters_only}{2}m/;
```

The *end of line string anchor* (`\Z`) requires that a match end at the end of a line within the string.

```
# also matches "loom", but an obvious improvement
my $seven_down = qr/\Al${letters_only}{2}m\Z/;
```

The *word boundary anchor* (`\b`) matches only at the boundary between a word character (`\w`) and a non-word character (`\W`). Use an anchored regex to find `loam` while prohibiting `Belgium`:

```
my $seven_down = qr/\bl${letters_only}{2}m\b/;
```

Metacharacters

Perl interprets several characters in regular expressions as *metacharacters*, characters represent something other than their literal interpretation. Metacharacters give regex wielders power far beyond mere substring matches. The regex engine treats all metacharacters as atoms.

The `.` metacharacter means "match any character except a newline". Remember that caveat; many novices forget it. A simple regex search--ignoring the obvious improvement of using anchors--for 7 Down might be `/l..m/`. Of course, there's always more than one way to get the right answer:

```
for my $word (@words)
{
    next unless length( $word ) == 4;
    next unless $word =~ /lB<..>m/;
    say "Possibility: $word";
}
```

If the potential matches in `@words` are more than the simplest English words, you will get false positives. `.` also matches punctuation characters, whitespace, and numbers. Be specific! The `\w` metacharacter represents all alphanumeric characters (*unicode*) and the underscore:

```
    next unless $word =~ /lB<\w\w>m/;
```

The `\d` metacharacter matches digits (also in the Unicode sense):

```
# not a robust phone number matcher
next unless $number =~ /B<\d>{3}-B<\d>{3}-B<\d>{4}/;
say "I have your number: $number";
```

Use the `\s` metacharacter to match whitespace, whether a literal space, a tab character, a carriage return, a form-feed, or a newline:

```
my $two_three_letter_words = qr/\w{3}B<\s>\w{3}/;
```

These metacharacters have negated forms. Use `\W` to match any character *except* a word character. Use `\D` to match a non-digit character. Use `\S` to match anything but whitespace. Use `\B` to match anywhere except a word boundary.

Character Classes

When none of those metacharacters is specific enough, specify your own *character class* by enclosing them in square brackets:

```
my $ascii_vowels = qr/B<[>aeiouB<]>/;
my $maybe_cat    = qr/c${ascii_vowels}t/;
```

Without those curly braces, Perl's parser would interpret the variable name as `$ascii_vowelst`, which either causes a compile-time error about an unknown variable or interpolates the contents of an existing `$ascii_vowelst` into the regex.

The hyphen character (`-`) allows you to specify a contiguous range of characters in a class, such as this `$ascii_letters_only` regex:

```
my $ascii_letters_only = qr/[a-zA-Z]/;
```

To include the hyphen as a member of the class, move it to the start or end:

```
my $interesting_punctuation = qr/[-!?]/;
```

... or escape it:

```
my $line_characters = qr/[|=\-_]/;
```

Use the caret (^) as the first element of the character class to mean "anything *except* these characters":

```
my $not_an_ascii_vowel = qr/[^aeiou]/;
```

Use a caret anywhere but the first position to make it a member of the character class. To include a hyphen in a negated character class, place it after the caret or at the end of the class, or escape it.

Capturing

Regular expressions allow you to group and capture portions of the match for later use. To extract an American telephone number of the form `(202) 456-1111` from a string:

```
my $area_code    = qr/\(\d{3}\)/;
my $local_number = qr/\d{3}-?\d{4}/;
my $phone_number = qr/$area_code\s?$local_number/;
```

Note especially the escaping of the parentheses within `$area_code`. Parentheses are special in Perl 5 regular expressions. They group atoms into larger units and also capture portions of matching strings. To match literal parentheses, escape them with backslashes as seen in `$area_code`.

Named Captures

Perl 5.10 added *named captures*, which allow you to capture portions of matches from applying a regular expression and access them later, such as finding a phone number in a string of contact information:

```
if ($contact_info =~ /(?<phone>$phone_number)/)
{
    say "Found a number $+{phone}";
}
```

Regexes tend to look like punctuation soup until you can group various portions together as chunks. Named capture syntax has the form:

```
(?<capture name> ... )
```

Parentheses enclose the capture. The `?< name >` construct names this particular capture and must immediately follow the left parenthesis. The remainder of the capture is a regular expression.

When a match against the enclosing pattern succeeds, Perl stores the portion of the string which

matches the enclosed pattern in the magic variable %+. In this hash, the key is the name of the capture and the value is the appropriate portion of the matched string.

Numbered Captures

Perl has supported *numbered captures* for ages:

```
if ($contact_info =~ /($phone_number)/)
{
    say "Found a number $1";
}
```

This form of capture provides no identifying name and does not store in %+. Instead, Perl stores the captured substring in a series of magic variables. The *first* matching capture that Perl finds goes into $1, the second into $2, and so on. Capture counts start at the *opening* parenthesis of the capture; thus the first left parenthesis begins the capture into $1, the second into $2, and so on.

While the syntax for named captures is longer than for numbered captures, it provides additional clarity. Counting left parentheses is tedious work, and combining regexes which each contain numbered captures is far too difficult. Named captures improve regex maintainability--though name collisions are possible, they're relatively infrequent. Minimize the risk by using named captures only in top-level regexes.

In list context, a regex match returns a list of captured substrings:

```
if (my ($number) = $contact_info =~ /($phone_number)/)
{
    say "Found a number $number";
}
```

Numbered captures are also useful in simple substitutions, where named captures may be more verbose:

```
my $order = 'Vegan brownies!';

$order =~ s/Vegan (\w+)/Vegetarian $1/;
# or
$order =~ s/Vegan (?<food>\w+)/Vegetarian $+{food}/;
```

Grouping and Alternation

Previous examples have all applied quantifiers to simple atoms. You may apply them to any regex element:

```
my $pork  = qr/pork/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork?.*?$beans/,
     'maybe pork, definitely beans' );
```

If you expand the regex manually, the results may surprise you:

```
my $pork_and_beans = qr/\Apork?.*beans/;

like( 'pork and beans', qr/$pork_and_beans/,
    'maybe pork, definitely beans' );
like( 'por and beans', qr/$pork_and_beans/,
```

```
        'wait... no phylloquinone here!' );
```

Sometimes specificity helps pattern accuracy:

```
my $pork  = qr/pork/;
my $and   = qr/and/;
my $beans = qr/beans/;

like( 'pork and beans', qr/\A$pork? $and? $beans/,
    'maybe pork, maybe and, definitely beans' );
```

Some regexes need to match either one thing or another. The *alternation* metacharacter (`|`) expresses this intent:

```
my $rice  = qr/rice/;
my $beans = qr/beans/;

like( 'rice',  qr/$rice|$beans/, 'Found rice'  );
like( 'beans', qr/$rice|$beans/, 'Found beans' );
```

The alternation metacharacter indicates that either preceding fragment may match. Keep in mind that alternation has a lower precedence (*precedence*) than even atoms:

```
like(   'rice',  qr/rice|beans/, 'Found rice'   );
like(   'beans', qr/rice|beans/, 'Found beans'  );
unlike( 'ricb',  qr/rice|beans/, 'Found hybrid' );
```

While it's easy to interpret `rice|beans` as meaning `ric`, followed by either `e` or `b`, followed by `eans`, alternations always include the *entire* fragment to the nearest regex delimiter, whether the start or end of the pattern, an enclosing parenthesis, another alternation character, or a square bracket.

To reduce confusion, use named fragments in variables (`$rice|$beans`) or group alternation candidates in *non-capturing groups*:

```
my $starches = qr/(?:pasta|potatoes|rice)/;
```

The `(?:)` sequence groups a series of atoms without making a capture.

A stringified regular expression includes an enclosing non-capturing group; `qr/rice|beans/` stringifies as `(?^u:rice|beans)`.

Other Escape Sequences

To match a *literal* instance of a metacharacter, *escape* it with a backslash (`\`). You've seen this before, where `\(` refers to a single left parenthesis and `\]` refers to a single right square bracket. `\.` refers to a literal period character instead of the "match anything but an explicit newline character" atom.

You will likely need to escape the alternation metacharacter (`|`) as well as the end of line metacharacter ($) and the quantifiers (+, ?, *).

The *metacharacter disabling characters* (`\Q` and `\E`) disable metacharacter interpretation within their boundaries. This is especially useful when taking match text from a source you don't control when writing the program:

```
my ($text, $literal_text) = @_;

return $text =~ /\Q$literal_text\E/;
```

The `$literal_text` argument can contain anything--the string `** ALERT **`, for example. Within the fragment bounded by `\Q` and `\E`, Perl interpret the regex as `\*\* ALERT \*\*` and attempt to match literal asterisk characters, rather than greedy quantifiers.

Be cautious when processing regular expressions from untrusted user input. A malicious regex master can craft a denial-of-service attack against your program.

Assertions

Regex anchors such as `\A`, `\b`, `\B`, and `\Z` are a form of *regex assertion*, which requires that the string meet some condition. These assertions do not match individual characters within the string. No matter what the string contains, the regex `qr/\A/` will *always* match..

Zero-width assertions match a *pattern*. Most importantly, they do not consume the portion of the pattern that they match. For example, to find a cat on its own, you might use a word boundary assertion:

```
my $just_a_cat = qr/cat\b/;
```

... but if you want to find a non-disastrous feline, you might use a *zero-width negative look-ahead assertion*:

```
my $safe_feline = qr/cat(?!astrophe)/;
```

The construct `(?!...)` matches the phrase `cat` only if the phrase `astrophe` does not immediately follow.

The *zero-width positive look-ahead assertion*:

```
my $disastrous_feline = qr/cat(?=astrophe)/;
```

... matches the phrase `cat` only if the phrase `astrophe` immediately follows. While a normal regular expression can accomplish the same thing, consider a regex to find all non-catastrophic words in the dictionary which start with `cat`:

```
my $disastrous_feline = qr/cat(?!astrophe)/;

while (<$words>)
{
    chomp;
    next unless /\A(?<cat>$disastrous_feline.*)\Z/;
    say "Found a non-catastrophe '$+{cat}'";
}
```

The zero-width assertion consumes none of the source string, leaving the anchored fragment <.*\Z>

to match. Otherwise, the capture would only capture the `cat` portion of the source string.

To assert that your feline never occurs at the start of a line, you might use a *zero-width negative look-behind assertion*. These assertions must have fixed sizes; you may not use quantifiers:

```
my $middle_cat = qr/(?<!\A)cat/;
```

The construct `(?<!...)` contains the fixed-width pattern. You could also express that the `cat` must always occur immediately after a space character with a *zero-width positive look-behind assertion*:

```
my $space_cat = qr/(?<=\s)cat/;
```

The construct `(?<=...)` contains the fixed-width pattern. This approach can be useful when combining a global regex match with the `\G` modifier, but it's an advanced feature you likely won't use often.

A newer feature of Perl 5 regexes is the *keep* assertion `\K`. This zero-width positive look-behind assertion *can* have a variable length:

```
my $spacey_cat = qr/\s+\Kcat/;

like( 'my cat has been to space', $spacey_cat );
like( 'my  cat  has  been  to  doublespace',
     $spacey_cat );
```

`\K` is surprisingly useful for certain substitutions which remove the end of a pattern:

```
my $exclamation = 'This is a catastrophe!';
$exclamation    =~ s/cat\K\w+!/./;

like( $exclamation, qr/\bcat\./,
                      "That wasn't so bad!" );
```

Regex Modifiers

Several modifiers change the behavior of the regular expression operators. These modifiers appear at the end of the match, substitution, and `qr//` operators. For example, to enable case-insensitive matching:

```
my $pet = 'CaMeLiA';

like( $pet, qr/Camelia/,  'Nice butterfly!'  );
like( $pet, qr/Camelia/i, 'shift key br0ken' );
```

The first `like()` will fail, because the strings contain different letters. The second `like()` will pass, because the `/i` modifier causes the regex to ignore case distinctions. `M` and `m` are equivalent in the second regex due to the modifier.

You may also embed regex modifiers within a pattern:

```
my $find_a_cat = qr/(?<feline>(?i)cat)/;
```

The `(?i)` syntax enables case-insensitive matching only for its enclosing group: in this case, the named capture. You may use multiple modifiers with this form. Disable specific modifiers by preceding them with the minus character (`-`):

```
my $find_a_rational = qr/(?<number>(?-i)Rat)/;
```

The multiline operator, `/m`, allows the `\A` and `\Z` anchors to match at any newline embedded within the string.

The `/s` modifier treats the source string as a single line such that the `.` metacharacter matches the newline character. Damian Conway suggests the mnemonic that `/m` modifies the behavior of *multiple* regex metacharacters, while `/s` modifies the behavior of a *single* regex metacharacter.

The `/r` modifier causes a substitution operation to return the result of the substitution, leaving the original string as-is. If the substitution succeeds, the result is a modified copy of the original. If the substitution fails (because the pattern does not match), the result is an unmodified copy of the original:

```
my $status     = 'I am hungry for pie.';
my $newstatus  = $status =~ s/pie/cake/r;
my $statuscopy = $status
               =~ s/liver and onions/bratwurst/r;

is( $status, 'I am hungry for pie.',
    'original string should be unmodified' );

like( $newstatus,    qr/cake/,      'cake wanted' );
unlike( $statuscopy, qr/bratwurst/, 'wurst not'   );
```

The `/x` modifier allows you to embed additional whitespace and comments within patterns. With this modifier in effect, the regex engine ignores whitespace and comments. The results are often much more readable:

```
my $attr_re = qr{
    \A                    # start of line

    (?:
      [;\n\s]*            # spaces and semicolons
      (?:/\*.*?\*/)?      # C comments
    )*

    ATTR

    \s+
    (   U?INTVAL
      | FLOATVAL
      | STRING\s+\*
    )
}x;
```

This regex isn't *simple*, but comments and whitespace improve its readability. Even if you compose

regexes together from compiled fragments, the /x modifier can still improve your code.

The /g modifier matches a regex globally throughout a string. This makes sense when used with a substitution:

```
# appease the Mitchell estate
my $contents = slurp( $file );
$contents    =~ s/Scarlett O'Hara/Mauve Midway/g;
```

When used with a match--not a substitution--the \G metacharacter allows you to process a string within a loop one chunk at a time. \G matches at the position where the most recent match ended. To process a poorly-encoded file full of American telephone numbers in logical chunks, you might write:

```
while ($contents =~ /\G(\w{3})(\w{3})(\w{4})/g)
{
    push @numbers, "($1) $2-$3";
}
```

Be aware that the \G anchor will take up at the last point in the string where the previous iteration of the match occurred. If the previous match ended with a greedy match such as .*, the next match will have less available string to match. Lookahead assertions can also help.

The /e modifier allows you to write arbitrary Perl 5 code on the right side of a substitution operation. If the match succeeds, the regex engine will run the code, using its return value as the substitution value. The earlier global substitution example could be simpler with code like:

```
# appease the Mitchell estate
$sequel  =~ s{Scarlett( O'Hara)?}
             {
                'Mauve' . defined $1
                        ? ' Midway'
                        : ''
             }ge;
```

Each additional occurrence of the /e modifier will cause another evaluation of the result of the expression, though only Perl golfers use anything beyond /ee.

Smart Matching

The smart match operator, ~~, compares two operands and returns a true value if they match. The fuzziness of the definition demonstrates the smartness of the operator: the type of comparison depends on the type of both operands. given (*given_when*) performs an implicit smart match.

The smart match operator is an infix operator:

```
say 'They match (somehow)' if $loperand ~~ $roperand;
```

The type of comparison generally depends first on the type of the right operand and then on the left operand. For example, if the right operand is a scalar with a numeric component, the comparison will use numeric equality. If the right operand is a regex, the comparison will use a grep or a pattern

match. If the right operand is an array, the comparison will perform a grep or a recursive smart match. If the right operand is a hash, the comparison will check the existence of one or more keys. A large and intimidating chart in `perldoc perlsyn` gives far more details about all the comparisons smart match can perform.

A serious proposal for 5.16 suggests simplifying smart match substantially. The more complex your operands, the more likely you are to receive confusing results. Avoid comparing objects and stick to simple operations between two scalars or one scalar and one aggregate for the best results.

With that said, smart match can be useful:

```
my ($x, $y) = (10, 20);
say 'Not equal numerically' unless $x ~~ $y;

my $z = '10 little endians';
say 'Equal numeric-ishally' if $x ~~ $z;

# regular expression match
my $needle = qr/needle/;

say 'Pattern match' if 'needle' ~~ $needle;

say 'Grep through array' if @haystack ~~ $needle;

say 'Grep through hash keys' if %hayhash ~~ $needle;

say 'Grep through array' if $needle ~~ @haystack;

say 'Array elements exist as hash keys'
    if %hayhash    ~~ @haystack;

say 'Smart match elements' if @straw ~~ @haystack;

say 'Grep through hash keys' if $needle ~~ %hayhash;

say 'Array elements exist as hash keys'
    if @haystack  ~~ %hayhash;

say 'Hash keys identical' if %hayhash ~~ %haymap;
```

Smart match works even if one operand is a *reference* to the given data type:

```
say 'Hash keys identical' if %hayhash ~~ \%hayhash;
```