• Perl designed to allow convenient processing of text.

• Perl syntax allows concise of regular expressions, e.g.

```perl
if ($name =~ /[0-9]$/) {
    # execute if regex match succeeds
    print "name ends with a digit\n";
}
```

$name = "Andrew";
# remove vowels from variable name
$name =~ s/[aeiou]///ig;
print "$name\n";  # prints ndrw
Perl extendeds POSIX regular expressions with some shorthand:

\d matches any digit, for ASCII: [0–9]
\D matches any non-digit, for ASCII: [^0–9]
\w matches any word char, for ASCII: [a-zA-Z_0-9]
\W matches any non-word char, for ASCII: [^a-zA-Z_0-9]
\s matches any whitespace, for ASCII: [\t\n\r\f]
\S matches any non-whitespace, for ASCII: [^ \t\n\r\f]
Perl Regular Expressions

Perl also adds some new anchors to regexps:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\b</td>
<td>matches at a word boundary</td>
</tr>
<tr>
<td>\B</td>
<td>matches except at a word boundary</td>
</tr>
</tbody>
</table>

And generalises the repetition operators:

<table>
<thead>
<tr>
<th>Pattern</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>patt*</td>
<td>matches 0 or more occurrences of \textit{patt}</td>
</tr>
<tr>
<td>patt+</td>
<td>matches 1 or more occurrences of \textit{patt}</td>
</tr>
<tr>
<td>patt?</td>
<td>matches 0 or 1 occurrence of \textit{patt}</td>
</tr>
<tr>
<td>patt{n}</td>
<td>matches \textit{n} occurrences of \textit{patt}</td>
</tr>
<tr>
<td>patt{n,}</td>
<td>matches at least \textit{n} occurrences of \textit{patt}</td>
</tr>
<tr>
<td>patt{n,m}</td>
<td>matches between \textit{n} and \textit{m} occurrences of \textit{patt}</td>
</tr>
</tbody>
</table>
Capturing What A Regex Matches

- after a regex match/substitute the variable `$&` is set to the matched string

```perl
$var = "pillow";
$var =~ /[aiou].*[aeiou]/;
print "$&\n"; # prints illo
```

- Brackets can be used to capture part of regex matches
- if a regex contains ()s the variables `$1`, `$2`, ... are set
- `$n` is set to the part of the string matched by the $n$th-pair of brackets

```perl
$var = "hello Andrew";
$var =~ /\(\w+)\s+\(\w+)\;/
print "1=$1 2=$2\n"; # prints "1=hello 2=Andrew"
```

- `$1` `$2`, ... are very useful in substitute operations, e.g.

```perl
$var = "The answer is 42 or 43?";
$var =~ s/.*(\d+).*/$1/;
print "$var\n"; # prints "42"
```
Greedy versus non-Greedy Pattern Matching

- The default semantics for pattern matching is **greedy**:  
  - starts match the first place it can succeed  
  - make the match as long as possible
- The ? operator changes pattern matching to **non-greedy**:  
  - starts match the first place it can succeed  
  - make the match as short as possible

$s = "abbabbbb";$
$s =~ s/ab+//;$
print "$s\n"; # prints "abbbb"

$s = "abbabbbb";$
$s =~ s/ab+?//;$
print "$s\n"; # prints "babbbb"
Why Implementing a Regex Matching isn’t Easy

- regex matching starts match the first place it can succeed
- but a regex can partly match many places

```perl
$s = "abbabbbbbbbabbbbc";
$s =~ s/ab+c//;
print "$s\n"; # prints "abbabbbbbbb"
```

- and may need to **backtrack**, e.g:

```perl
$s = "abbabbbbbbbcabbb";
$s =~ s/a.*bc//;
print "$s\n"; # prints "abbb"
```

- poorly design regex engines can get very slow
  - have been used for denial-of-service attacks
- Perl extensions (back-references) make matching NP-hard
Perl allows variables to be interpolated to regexes, e.g:

```perl
$pattern = "ab+";
$replace = "Yod";
$text = "abba";

$text =~ s/$pattern/$replace/;
print "$text\n"; prints "Yoda"
```

Note: Perl doesn’t confuse the use of $ in /$var/ and /abc$/., because the anchor $ occurs at the end of the regex.
In a **scalar** context, Perl matching & substitute operators return how many times the match/substitute succeeded.

This is convenient for control of if/while statements, e.g:

```perl
print "Destroy the file system? ";
$answer = <STDIN>;
if ($answer =~ /yes|ok|affirmative/i) {
    system "rm -r /";
}

$var =~ s/[aeiou]//ig or die "no vowels to replace";
```
Matching Operators in a List Context

In a list context, matching operators returns a list of the matched strings, e.g:

```
$string = "-5==10zzz200_";
@numbers = $string =~ /\d+/g;
print join"," @numbers), "\n"; # prints 5,10,200
```

If the regex contains () the captured text is returned

```
$string = "Hopper, Grace Brewster Murray";
($family_name, $given_name) = $string =~ /([^-],)*, (\S+)/;
print "$given_name $family_name\n"; # prints Grace Hopper
```

Simple Perl script to accept a pattern and a string and show the match (if any):
#!/usr/bin/perl

$pattern = $ARGV[0]; print "pattern=/$pattern/\n";
$string = $ARGV[1]; print "string ="$string"\n";
$string =~ /$pattern/; print "match ="$&"\n";

- fantastic website for exploring and understanding regexs:
  https://regex101.com/ or https://regexr.com/
# For each file given as argument replace occurrences of Hermione
# with Harry and vice-versa, relies on file not containing Zaphod
# modified text is written to a temporary file

```perl
foreach $filename (@ARGV) {
    $tmp_filename = "$filename.new";
    die "$0: $tmp_filename already exists" if -e "$tmp_filename";
    open my $f, '<', $filename or die "$0: Can not open $filename: $!";
    open my $g, '>', $tmp_filename or die "$0: Can not open $tmp_filename : $!";
    while ($line = <$f>) {
        $line =~ s/Herm[io]+ne/Zaphod/g;
        $line =~ s/Harry/Hermione/g;
        $line =~ s/Zaphod/Harry/g;
        print $g $line;
    }
    close $f;
    close $g;
    rename "$tmp_filename", $filename or die "$0: Can not rename file";
}
```

source code for gender_reversal.0.pl
# For each file given as argument replace occurrences of Hermione with Harry and vice-versa, relies on file not containing Zaphod modified text is stored in array then file over-written

```perl
foreach $filename (@ARGV) {
    open my $f, '<', $filename or die "$0: Can not open $filename: $!";
    $line_count = 0;
    while ($line = <$f>) {
        $line =~ s/Herm[io]+ne/Zaphod/g;
        $line =~ s/Harry/Hermione/g;
        $line =~ s/Zaphod/Harry/g;
        $new_lines[$line_count++] = $line;
    }
    close $f;
    open my $g, '>', "$filename"
        or die "$0: Can not open $filename : $!";
    print $g @new_lines;
    close $g;
}
```

source code for gender_reversal.1.pl
Example - Gender Swap #2

For each file given as argument replace occurrences of Hermione with Harry and vice-versa, relies on file not containing Zaphod
modified text is stored in array then file over-written

```perl
foreach $filename (@ARGV) {
    open my $f, '<', $filename or
die "$0: Can not open $filename: $!";
@lines = <$f>;
close $f;
# note loop variable $line is aliased to array elements
# changes to it change the corresponding array element
foreach $line (@lines) {
    $line =~ s/Hermio+/Zaphod/g;
    $line =~ s/Harry/Hermione/g;
    $line =~ s/Zaphod/Harry/g;
}
    open my $g, '>', "$filename" or
die "$0: Can not open $filename : $!";
print $g @lines;
close $g;
}
```

source code for gender_reversal.2.pl
Example - Gender Swap #3

# For each file given as argument replace occurrences of Hermione
# with Harry and vice-versa, relies on file not containing Zaphod
# modified text is stored in array then file over-written
# See http://www.perlmonks.org/?node_id=1952
# for alternative way to read a file into a string

foreach $filename (@ARGV) {
    open my $f, '<', $filename or
die "$0: Can not open $filename: $!";

    while ($line = <$f>) {
        $novel .= $line;
    }

    close $f;

    $novel =~ s/Herm[io]+ne/Zaphod/g;
    $novel =~ s/Harry/Hermione/g;
    $novel =~ s/Zaphod/Harry/g;

    open my $g, '>', "$filename" or
die "$0: Can not open $filename : $!";

    print $g $novel;

    close $g;
}

source code for gender_reversal.3.pl
```bash
#!/usr/bin/perl -w -i
# written by andrewt@cse.unsw.edu.au as a COMP2041 example
# For each file given as argument replace occurrences of Hermione
# with Harry and vice-versa, relies on file not containing Zaphod
# The unix filter-like behaviour of <> is used to read files
# Perl's -i option replaces file with output from script
while ($line = <>) {
    chomp $line;
    $line =~ s/Herm\[io\]+ne/Zaphod/g;
    $line =~ s/Harry/Hermione/g;
    $line =~ s/Zaphod/Harry/g;
    print $line;
}
```

source code for gender_reversal.4.pl
#!/usr/bin/perl -w -i
# written by andrewt@cse.unsw.edu.au as a COMP2041 example
# For each file given as argument replace occurrences of Hermione
# with Harry and vice-versa, relies on file not containing Zaphod
# The unix filter-like behaviour of <> is used to read files
# Perl's -i option replaces file with output from script
# Perl's default variable $ _ is used
while (<>)
{
   s/Herm[io]+ne/Zaphod/g;
   s/Harry/Hermione/g;
   s/Zaphod/Harry/g;
}

source code for gender_reversal.5.pl
#!/usr/bin/perl -w -p -i
# written by andrewt@cse.unsw.edu.au as a COMP2041 example
# For each file given as argument replace occurrences of Hermione
# with Harry and vice-versa, relies on file not containing Zaphod
# The unix filter-like behaviour of  <> is used to read files
# Perl's -i option replaces file with output from script
# Perl's -p option is used to produce unix filter-like behaviour.
# Perl's default variable $_ is used
s/Herm[io]+ne/Zaphod/g;
s/Harry/Hermione/g;
s/Zaphod/Harry/g;

source code for gender_reversal.6.pl
Example - fetching a web page #0

# Fetch a web page removing HTML tags and constants (e.g &)
# Lines between script or style tags are skipped.
# Non-blank lines are printed
# There are better ways to fetch web pages (e.g. HTTP::Request::Common)
# The regex code below doesn't handle a number of cases. It is often
# better to use a library to properly parse HTML before processing it.
# But beware illegal HTML is common & can break parsers.

```perl
foreach $url (@ARGV) {
    open my $f, '-|', "wget -q -O- "$url"" or die;
    while ($line = <$f>) {
        if ($line =~ /^\s*<(script|style)/i) {
            while ($line = <$f>) {
                last if $line =~ /^\s*\</(script|style)/i;
            }
        } else {
            $line =~ s/&\w+/ /g;
            $line =~ s/<[^>]*>///g;
            print $line if $line =~ \S/;
        }
    }
    close $f;
}
```

source code for wget0.pl
# Fetch a web page removing HTML tags and constants
# The contents of script or style tags are removed..
# Non-blank lines are printed
#
# The regex code below doesn't handle a number of cases. It is often
# better to use a library to properly parse HTML before processing it.
# But beware illegal HTML is common & often causes problems for parsers.
#
# note the use of the s modifier to allow . to match a newline
#
use LWP::Simple;
foreach $url (@ARGV) {
    $html = get $url;
    # remove script tags including contents
    $html =~ s/<script.*?<\/script>//isg;
    # remove style tags including contents
    $html =~ s/<style.*?<\/style>//isg;
    # remove tags
    $html =~ s/<.*?>//isg;
    # remove blank lines
    $html =~ s/\n\s*\n/\n/ig;
    print $html;
}
# Find the positive integers among input text
# print their sum and mean
# Note regexp to split on non-digits
# Note check to handle empty string from split

@input_text_array = <>;
$input_text_array = join "", @input_text_array;
@numbers = split(/\D+/, $input_text_array);
print join("", @numbers), "\n";
foreach $number (@numbers) {
    if ($number ne '') {
        $total += $number;
        $n++;
    }
}
if (@numbers) {
    printf "%s numbers: total $total mean %s\n", $total/$n;
}
# Find integers (positive and negative) among input text
# print their sum and mean
# Note regexp to match number: -?\d+
# Harder to use split here (unlike just positive integers)

@input_text_array = <>;
$input_text_array = join "", @input_text_array;
@numbers = $input_text_array =~ /-?\d+/g;

foreach $number (@numbers) {
    $total += $number;
}

if (@numbers) {
    $n = @numbers;
    printf "$n numbers: total $total mean %s
", $total/$n;
}

source code for find_numbers.1.pl
# Print the last number (real or integer) on every line
# Note regexp to match number:  -?\d+(\./\d+)?

while ($line = <>) {
    if ($line =~ /(-?\d+(\./\d+)?)\D*$/) {
        print "$1\n";
    }
}

code for print_last_number.pl