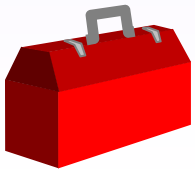
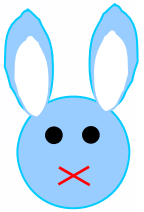
Overview of Perl Basics

Just enough to help you read a Perl program

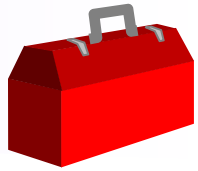
(draft v0.5)

Isaac Lin



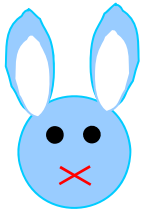


Types



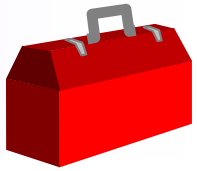
- **Three basic types:**
  - scalar: number, string, or reference
    13, “bunny”
  - list/array of scalars
    (1, 2, “carrot”, 26)
  - hash: associative array
- **Simple variable definition with initialization**
  - $a = 13;
  - @a = (“carrot”, “celery”, “lettuce”);
  - %a = (carrot => “orange”, celery => “green”);

    => is the same as a comma (,), except that it implicitly quotes the string on the left-hand side of the =>

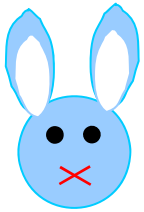


- **Note $a, @a, and %a are all different variables.**

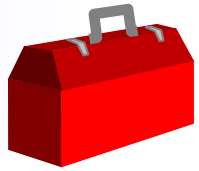
Common Perl features, part 1



- **all C operators supported**
- **string comparison operators:**
  - eq: equal to
  - ne: not equal to
  - lt: less than
  - le: less than or equal
  - gt: greater than
  - ge: greater than or equal
  - cmp: -1 if LHS is less than RHS, 1 if RHS is greater than LHS, 0 if they are equal
    - <=> is the equivalent numeric comparison operator
  - string comparisons use ASCII or Unicode sorting order
- **concatenation operator**
  - “bunny” . “ rabbit” eq “bunny rabbit”

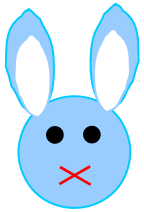


Conversion of scalars

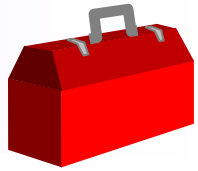


- **Numeric and string scalars are converted upon need.**
  - If $a is used in a mathematical expression and $a is a string, then $a is converted to a number as required.
  - If $a is used where a string is expected, and $a is a number, then $a is converted to a string as required.
  - Examples:

```
my $a = "5";
my $b = $a + 2;          # $b == 7
my $c = $b . " steps";   # $c eq "7 steps"
```

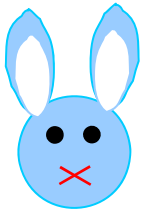


The Sigil (a Perl oddity)

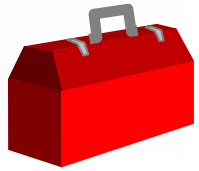


- **The “sigil” ($, @) is used to determine the return value, *not* the variable type.**
  - a[3] accesses index 3 in the @a list, but you must add a sigil to determine what type you want the return value to be.
    Examples:
    - $b = $a[3];
    - @b = @a[3]; # Equivalent to @b = ($a[3]);
  - a{"carrot"} accesses associative index “carrot” in the %a hash.
    Examples:
    - $b = $a{"carrot"}; OR $b = $a{carrot};
    - @b = @a{carrot}; # Equivalent to @b = ($a{carrot});
  - Usually, if there is just one index within the [] or {}, you intend to use a leading $ as the sigil.

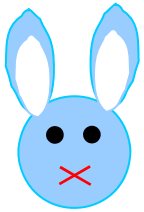
within {}, text is auto-quoted



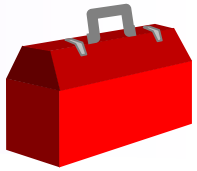
Assigning to lists of variables



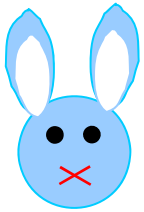
- **Example of equivalent expressions:**
  - ($a, $b, $c) = ($a[0], $a[1], $a[2]);
  - ($a, $b, $c) = @a[0, 1, 2];
  - ($a, $b, $c) = @a[0..2];



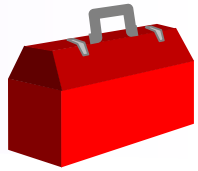
Scope



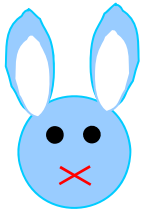
- **Variables defined with the “my” keyword are local to the current scope.**
  - If not within a block (i.e. not within { }), then the current scope is the file, and the variable is local to the file. It can be accessed from any scope in the file, but not from other files.
  - Examples:
    - my $a = "blue";
    - my @a = ("red", "green", "blue");
- **Variables defined without the “my” keyword are in the global scope.**
  - The variable is accessible from any scope in the file, and from other files.



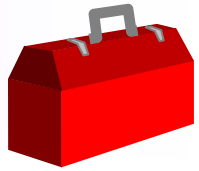
Scalar and list context



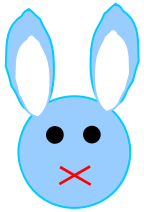
- **Expressions are evaluated in either a scalar or list context, and the return value may be different depending on context.**
  - @b = @a;
    - Since the lvalue is a list, @a is evaluated in list context, which returns the entire list. The list is then assigned to @b.
  - $listLength = @a;
    - Since the lvalue is a scalar, @a is evaluated in scalar context, which returns the length of the list.
  - @hashContents = %a;
    - %a is evaluated in list context, which returns a flat list of (key1, value1, key2, value2, ...). This list can be assigned to another hash, thereby copying the hash.
  - Beware: Many standard Perl functions and operators return different values in scalar or list context.
  - Scalar context can be forced using the scalar keyword.
    - ($listLength) = scalar @a;



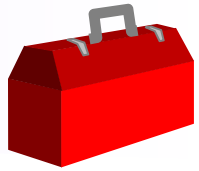
Boolean values



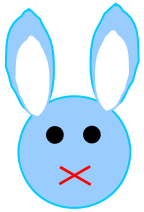
- **False values:**
  - 0, empty string (""), and undefined values (undef).
  - Uninitialized variables evaluate to undef.
  - If no value has been assigned to a given list or hash index, then an expression that accesses the list or hash using that index will evaluate to undef.
- **True values:**
  - All other values evaluate to true.



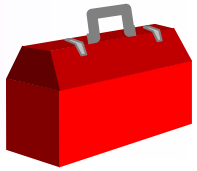
Quoting rules



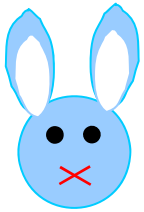
- **Double quotes:**
  - Variables within double quotes are interpolated.
  - Escape sequences are expanded. Examples:
    - \": "
    - \n: newline
    - \r: carriage return
    - \t: tab
    - \x*ab*: hexadecimal value *ab* is inserted
    - \x{*abcd*}: Unicode code point *abcd* is inserted
- **Single quotes:**
  - Only two escape sequences are expanded:
    - \\: \
    - \': '
  - No other expansion or interpolation is done.



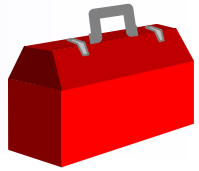
Quote operators



- **Operators:**
  - qq/*contents of string*/: same as "*contents of string* "
  - q/*contents of string*/: same as '*contents of string* '
  - qw/*word1 word2 word3*/: same as ('word1', 'word2', 'word3')
  - Note the / can be replaced by any non-alphanumeric, non-whitespace character. If (, [, <, or { is used as the first character, then the matching closing character must be used as the second character.
    - The delimiter character must be escaped within the string; the usual delimiter (' or ") does not .
- **Command execution (backtick) operator**
  - `command arg1 arg2`: executes command line, and the results from stdout are interpolated.
  - qx/*contents of string*/: same as `*contents of string*`
  - Example: my $dirListing = `ls -alF`;



Subroutines

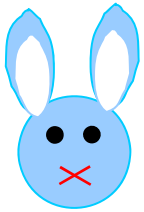


- **If the subroutine is defined before use:**
  - Can be invoked using *sub_name*(*parameters*) or just *sub_name parameters*.
  - Invocations that appear before the definition: &*sub_name*(*parameters*).
- **Parameters passed to the subroutine are accessible within the subroutine using the @_ list.**
  - Each element (e.g. $_[0]) is an alias for the variable passed in, so if you modify it, you modify the original variable.

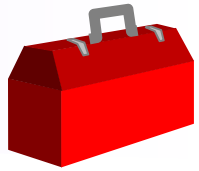
Examples:

```
sub increment {
  ++$_[0];
}  # sub increment
sub greetPerson {
  my ($name) = @_;
  print "Hello, ", $name, "!\n";
}
```

For legibility, if you do not need to modify the original variable, you should copy the parameters to a local variable.



Subroutine return values



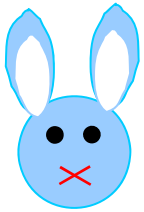
- **Subroutines can return a scalar or list. Hashes cannot be returned.**
  - If a hash is used in a return statement, the hash will be evaluated in list context. A flat list with the contents of the hash will be returned.
    Example:

```
sub showColourMap {
  return %globalColourMap;
}
```

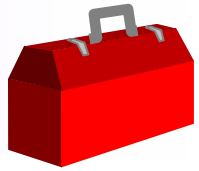
  - Note a reference is a scalar, and so references can be returned.
    Example:

```
sub cloneColourMap {
  return { %globalColourMap };
}
```

%globalColourMap is evaluated in list context, returning a flat list, and this list is used to create an anonymous hash. The anonymous hash is a copy of %globalColourMap.

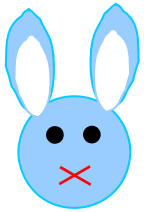


Control structures, part 1

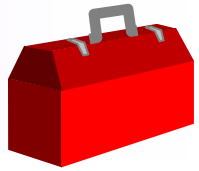


- **Similar to C, but with some additions:**
  - for (*init-expr*; *continue-expr*; *end-of-loop-expr*) { ... }
  - while (*expr*) { ... }
  - if (*expr*) { ... } elsif (*expr*) { ... } else { ... }
  - unless (*expr*) { ... } else { ... }
    - same as if (!*expr*) { ... } else { ... }
  - foreach *scalar-variable* (*list*) { ... }
    - foreach $item (@shoppingList) { ... }
    - foreach my $book ('War and Peace', 'A Tale of Two Cities') { ... }
    - The for and foreach keywords are interchangeable. For clarity, for is used for traditional for loops, and foreach is used for iterating through lists
  - do { ... }
    - Executes the statement block once (but see part 2 for other possibilities).
  - Note the {}'s are mandatory, unlike C.

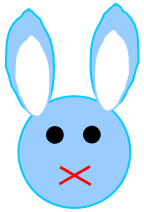
See "Common Perl features, part 3" for more on foreach



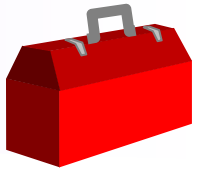
Control structures, part 2



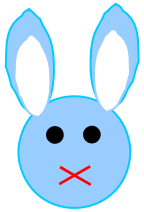
- **All conditional control structures can be used as a trailing modifier to a statement.**
  - $maxValue = $a if $a > $maxValue;
    - same as if ($a > $maxValue) { $maxValue = $a; }
  - do { *multiple statements* } while ($a <= $maxIterations);
    - same as while ($a<=$maxIterations) { *multiple statements* }
  - print $char foreach my $char (@characters);
    - same as foreach my $char (@characters) { print $char; }
  - Note () around the expression is optional when the control structure is used as a trailing modifier.



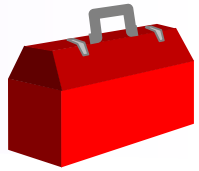
Control structures, part 3



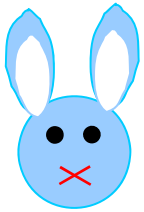
- **Shortcuts for if and unless statements**
  - doStuff($a) or print "Failed\n";
    - same as unless (doStuff($a)) { print "Failed\n"; } except that no value is returned by the unless statement
  - openBag($bag) and packStuff($bag);
    - same as if (openBag($bag)) { packStuff($bag); } except that no value is returned by the if statement
  - Older code may use || and &&, but because of problems related to operator precedence, use or and and instead (they have lowest precedence).
  - Continue to use ||, &&, and ! in expressions. Because or and and have lowest precedence, using them in expressions can cause confusion.



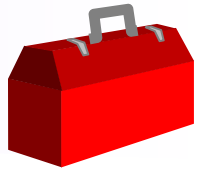
References



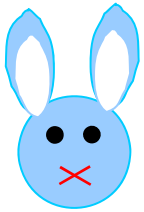
- **References to other variables**
  - $scalarRef = \$a;
  - $listRef = \@a;
  - $hashRef = \%a;
- **References to anonymous lists or hashes**
  - $listRef = ["my", "happy", "bunny"];
  - $hashRef = { carrot => "tasty", "rice cakes" => "dry" };
- **References to subroutines:**
  - $subRef = \&increment;
- **References to anonymous subroutines**
  - $subRef = sub { print "What's up?\n"; };



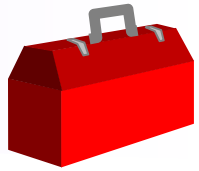
De-referencing references



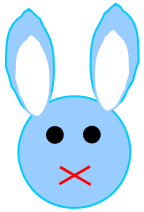
- **Enclose the reference within {} and add the appropriate prefix ($, @, %):**
  - Scalar reference: ${$scalarRef}
  - List reference: @{$listRef}
  - Hash reference: %{$hashRef}
- **To dereference a list or hash and index it, use the -> operator.**
  - $a = $listRef->[3];
  - $a = $hashRef->{carrot};
  - If you are accessing nested references, the -> can be omitted between][or }{.
    - $a = $listOfLists->[3][4]; # same as $listOfLists->[3]->[4]
    - $a = $hashOfHashes->{carrot}{price};
      # same as $hashOfHashes->{carrot}->{price}



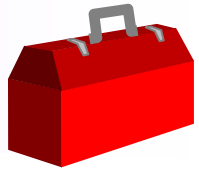
Packages



- **Naming conventions:**
  - A file defining a Perl package has a .pm extension.
  - The package name can be hierarchical, such as XML::Parser::Lite.
    - This maps to XML/Parser/Lite.pm. (i.e. the XML subdirectory + the Parser subdirectory + the Lite.pm file).
    - A Perl script includes a package as follows:
      use XML::Parser::Lite;
    - The Perl compiler will search the library search path for XML/Parser/Lite.pm.
    - The Perl script can add to the library search path using the use lib directive:
      use lib '/opt/projects/MyProj/lib/perl';



Creating and using a package



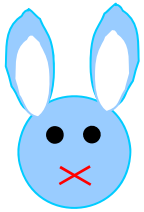
- **Grocery/Vegetable/Carrot.pm:**

```
package Grocery::Vegetable::Carrot;

# ... Contents of your package go here
sub countCarrots { ... }

1;   # The last statement in the file must return a true
     # value.
```

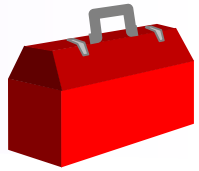
- **Invoking subroutines from a package:**

```
# User code
use Grocery::Vegetable::Carrot;

my $carrotCount = Grocery::Vegetable::Carrot::countCarrots();
```

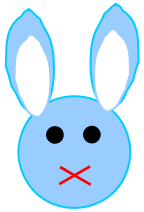
  - Packages can provide an import() subroutine that exports subroutines into the user's scope, so that countCarrots() can be called without having to qualify it with the package name.
    - See the Exporter module man page for more information.



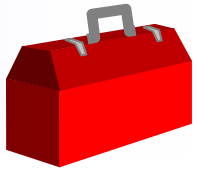
Common Perl features, part 2



- **push and pop operators**
  - push @a, "cucumber";
    - Adds "cucumber" to @a as the last element.
  - pop @a;
    - Removes the last element from @a and returns it.
- **shift and unshift operator.**
  - $firstElem = shift @a;
    - Removes a[0] from the list and shifts down all of the following elements (a[1] is moved to a[0], etc.).
    - a[0] is returned.
    - If no variable is specified as an argument to shift, then the @_ list is used.
    - Commonly used within subroutines to process the argument list.
  - unshift @a, "rabbit";
    - Shifts up all elements in @a (a[0] is moved to a[1], etc.) and adds "rabbit" to @a as the first element.



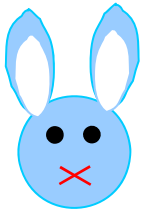
Classes



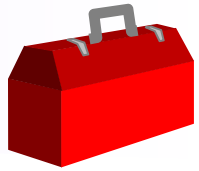
- **A Class is a package that provides a constructor.**
- **A constructor is a subroutine in the package that returns a blessed reference.**

```
package Grocery;
sub new {
  my $class = shift;
  my $self = {};
  %{$self} = @_;
  return bless $self, $class;
}
1;
```

- **Invoked as follows:**
  - my $obj = Grocery->new(name => "Sam's");
  - The class name is automatically passed in as the first argument.



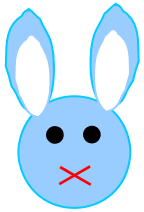
Methods



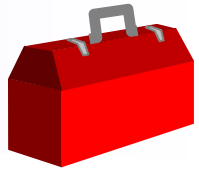
- **All subroutines within the package for the class are methods.**

```
package Grocery;
# ...
sub displaySign {
  my ($self,$tagline) = @_;
  print $self->{name}, " Grocery\n", $tagline, "\n";
}
# ...
# User code:
$obj->displaySign("Free cookies!");
```

  - The object reference is automatically passed in as the first parameter.



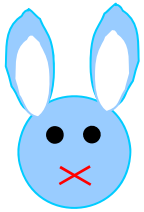
Inheritance of methods



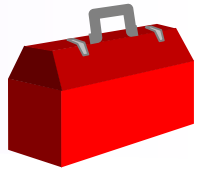
- **The @ISA global variable is used to specify the superclasses for a class.**

```
package Grocery::Vegetable::Carrot;
push @ISA, 'Grocery::Vegetable';
```

  - When a method is invoked on an object, if a corresponding subroutine is not found within the object's class, then each class in the @ISA list is searched in order, until a match is found.
  - Usually inheritance follows the package hierarchy (e.g. Grocery::Vegetable::Carrot inherits from Grocery::Vegetable). However, it does not have to.
    - In this example, Vegetable is *not* a Grocery; putting the Vegetable package under the Grocery package is just a way to organize the packages, and not the class hierarchy.



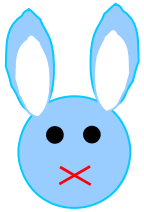
File and directory handles



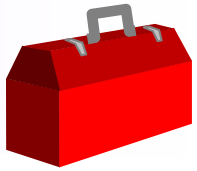
- **Uninitialized variables defined with the my keyword can be used as file or directory handles.**

```
{
  my $fh;
  open $fh, '<', 'some_file';  # open for reading
  close $fh;
}
```

  - The file handle will be closed automatically when $fh goes out of scope, if you forget to do it.
  - To open a file for writing, use '>' as the second argument.
  - See the perlfunc man page for more choices for the second argument.



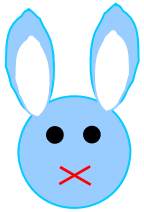
I/O



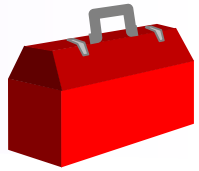
- **Writing to a filehandle**
  - print $fh *arg1*, *arg2*, *arg3*;

Note there is no comma between the file handle and the first argument.

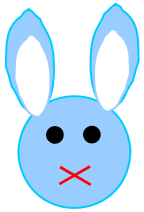
  - If the handle is omitted, output goes to STDOUT
    - print *arg1*, *arg2*, *arg3*;
  - Printing to stderr: print STDERR *arg1*, *arg2*, *arg3*;
- **Reading from a file using line input (angle) operator**
  - Scalar context: my $line = <$fh> reads one line from the file.
  - List context: my @lines = <$fh> reads in all lines, placing one line in each list element.
  - When the end of file is reached, <$fh> returns undef.



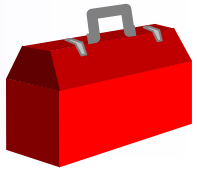
Common Perl features, part 3



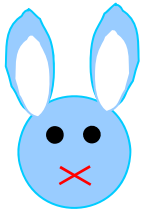
- **while (<$fh>) { ... }**
  - By default, each line read in from $fh is stored in $_.
  - When the end of file is reached, <$fh> will return undef, and the while loop will be exited.
  - To avoid problems with nested loops, it is preferable to use an explicit assignment:
    ```
    while (my $line = <$fh>) { ... }
    ```
- **foreach (*some_list*) { ... }**
  - Assigns each element of the list to $_ and executes the loop body.
  - $_ is an alias for the original value, so modifying $_ will modify the value.
  - foreach my $val (*some_list*) can be used to avoid implicit assignment to $_.
  - Examples: foreach my $val (@a) { ... },
    foreach my $line (<$fh>) { ... }
    foreach my $val ($a, $b, $c) { ... }



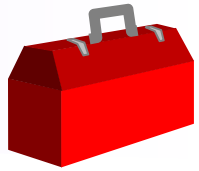
Common Perl features, part 4



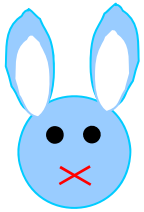
- **file test operators**
  - Similar to the file test operations available in the Unix test command.
  - -f *filename*: File is an ordinary file.
  - -r *filename*: File is readable.
  - -w *filename*: File is writeable.
  - -x *filename*: File is executable.
  - -e *filename*: File exists.
  - -z *filename*: File is zero size.
  - -s *filename*: File is non-zero size (returns size).
  - -d *filename*: File is actually a directory.
  - -l *filename*: File is a symbolic link.



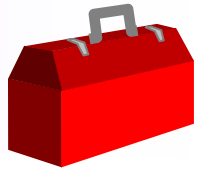
Regular Expressions, part 1



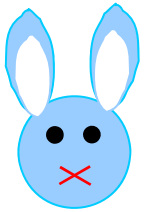
- **A regular expression is a pattern for matching strings.**
- **Basic syntax (see perlre man page for more details):**
  - Special metacharacters: . * + ? ( ) | [{ ^ $ \
  - Variables within the regular expression are interpolated. Thus @ and % are also special characters.
  - All other characters match directly.
  - . matches any character
  - foo|bar matches either foo *or* bar
  - [abcd] matches either a, b, c, or d
  - [^abcd] matches any character other than a, b, c, or d
  - ^ at the start of the pattern means that the pattern must start matching from the beginning of the string.
  - $ at the end of the pattern means that the pattern must finish matching at the end of the string.



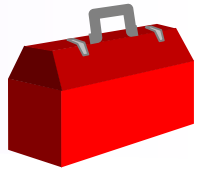
Regular Expressions, part 2



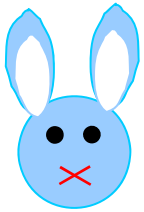
- Quantifiers:
  - a* matches zero or more occurrences of a
  - a+ matches one or more occurrences of a
  - a? matches zero or one occurrences of a
  - a{1,3} matches one, two, or three occurrences of a
  - a{3} matches aaa
  - () are used for grouping. Example: foo(bar)* matches foo followed by zero or more occurrences of bar
  - Quantifier matching is greedy. From left to right, as much as possible of each quantifier is matched, as long as the entire pattern can still match.
    - e.g. if matching (foo)?(foo){1,2}(foo)+(foo)* against “foofoofoofoofoo”, (foo)? matches the first foo, (foo){1,2} matches the second and third foo’s, (foo)+ matches the fourth and fifth foo’s, and (foo)* matches zero foo’s.



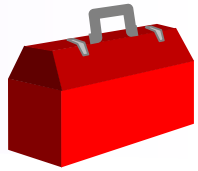
Regular Expressions, part 3



- \s matches any whitespace character (e.g space, tab, newline). \S matches any non-whitespace character.
- \d matches any digit character. \D matches any non-digit character.
- \w matches alphanumeric characters and _ (“word” characters). \W matches any non-word character.
- \b matches a word boundary (transition from word character to non-word character, or vice versa).
- \ followed by a non-alphanumeric character (including all special characters): matches that character
- \x{*abcd*} matches hexadecimal character *abcd*.
- \Q is a special instruction: any following metacharacters in the pattern up to \E are automatically escaped (“quoted”).

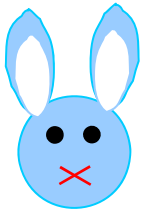


Matching with regular expressions

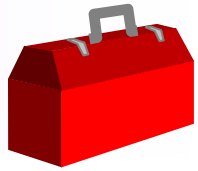


- **The string to be matched is bound to a matching operator.**
  - The expression evaluates to true if a match is found.
  - Traditional syntax: $a =~ /*some_pattern*/
  - Alternate syntax: $a =~ m/*some_pattern*/
    - With the alternate syntax, any non-alphanumeric, non-whitespace character can be used instead of /. e.g. $a =~ m!*some_pattern*!
    - If (, [, <, or { is used as the first character, then the matching closing character must be used as the second character. e.g. $a =~ m(*some_pattern*)
    - The alternate syntax is especially useful when the pattern contains /.
  - Examples:

```
print "My favourite foods!\n" if
  ($a =~ /^(pizza|steak)$/);
my $fUnderOptDir = $a =~ m,^/opt/,;
```



Capturing matches

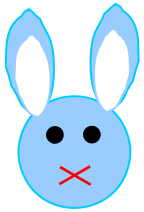


- **Any subpatterns grouped with () are captured in $1, $2, etc.**
  - The subpatterns are numbered by the position of the corresponding opening parenthesis (.
  - Example:

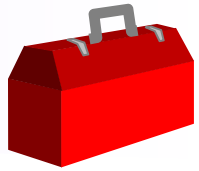
```
"mississippi" =~ /((iss)*ipp)/;
# $1 = "ississipp" and $2 = "iss"
```

  - To disable capturing for a group, use (?: ...)

```
"mississippi" =~ /((?:iss)*ipp)/;
# $1 = "ississipp", $2 is not set by the match
```



Substitutions with regular expressions

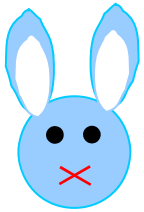


- **The substitution operator is used to find a matching pattern and replace it.**
  - The expression evaluates to true if a match is found.
  - Traditional syntax: $a =~ s/*pattern*/*replacement*/
  - Alternate syntax: Any non-alphanumeric, non-whitespace character can be used instead of /. If (, [, <, or { is used as the first character, then the matching closing character must be used as the second character, and another pair of characters is used to surround *replacement*. e.g. s{cake}<cookies>, s[cookies][candy]
  - Examples:

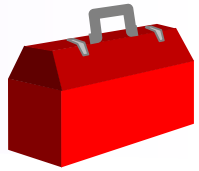
```
$a =~ s/\bgreat\b/stupendous/g;
$a =~ s,\bmug(s)?\b,cup$1,g;
```

    - Without the /g (“global”) modifier at the end, after one match, processing stops. With the /g modifier, after any match, the string is processed again, starting from the end of the previous match.
    - Note how captured matches can be used in the replacement pattern

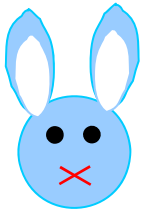
```
$a =~ s,^[Ff]irst,1st,;
```



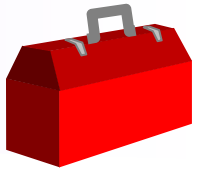
Backreferences within the pattern



- **Backreferences can be used to reference part of the pattern that had matched earlier.**
  - Example: $a =~ /Take the (boy|girl) to the \1s room\./



More information



- **ActiveState Perl documentation**
  - http://aspn.activestate.com/ASPN/docs/ActivePerl
- **Comprehensive Perl Archive Network (CPAN)**
  - http://www.cpan.org/
- **CBM/SDM Perl coding guidelines**
  - http://cbmproduct.ca.nortel.com/ -> Programming Model -> Application Guidelines -> Coding Guidelines: Perl
  - Note the Programming Model -> Programming References -> Perl page has links to the ActiveState documentation and CPAN.