Announcements

• Schedule posted on OnCourse
  – Click the Wiki link
  – Readings, resources, assignments posted here

Perl Roadmap

• We have covered:
  – I/O
  – scalars
  – arrays & lists
  – control flow

Today we’ll cover:

  – More on arrays & lists
  – Hashes
  – Regular expressions
  – Subroutines
  – File I/O
  – string & list manipulation

Later in the semester:

  – References
  – Modules

Data types in Perl

• Scalar types
  – Numbers, strings
  – variable names must start with $

• Lists (aka arrays)
  – Store a collection of scalars
  – variable names must start with @

• Hashes (aka associative arrays)
  – Store a mapping from scalars to scalars
  – variable names must start with %

Arrays and lists

• An array is another type of variable that holds an ordered list of scalar values
  – Array variable names begin with @

• Lists are written as comma-separated values, surrounded by parentheses, e.g.:

```perl
@names = ("Fil", "Dennis"); # 2 values
@ages = (45, 57, 22, 18); # 4 values
@userids = (); # 0 values
@ints = (3..7); # shortcut for (3,4,5,6,7)
```

Accessing array elements

• The first element of array @names is referred to as @names[0]
  – Array indices are 0-based, so the n-th element is @names[n-1]

```perl
my @names = ('b', 'obama', 'g', 'bush', 'b', 'clinton');
print @names[1], ' ', @names[3], ' ', @names[5], '
;
print "\n";
print (\$names[1], ' ', \$names[3], ' ', \$names[5], '\n');
print \n;
```

Arrays and lists

• An array is another type of variable that holds an ordered list of scalar values
  – Array variable names begin with @

• Lists are written as comma-separated values, surrounded by parentheses, e.g.:

```perl
@names = ("Fil", "Dennis"); # 2 values
@ages = (45, 57, 22, 18); # 4 values
@userids = (); # 0 values
@ints = (3..7); # shortcut for (3,4,5,6,7)
```

Accessing array elements

• The first element of array @names is referred to as @names[0]
  – Array indices are 0-based, so the n-th element is @names[n-1]

```perl
my @names = ('b', 'obama', 'g', 'bush', 'b', 'clinton');
print @names[1], ' ', @names[3], ' ', @names[5], '
;
print "\n";
print (\$names[1], ' ', \$names[3], ' ', \$names[5], '\n');
print \n;
```
More on arrays

- Find the index of the last element of @array using $#array
- Assigning elements past the end of an array automatically expands the size of the array

```perl
my @numbers = (1..9);
print $numbers[$#numbers] . "n";
print $numbers[9] . "n";
print $numbers[9+10] . "n";

10
```

Adding & removing elements

- push adds an element to the end of an array
- pop removes and returns the last element
- unshift adds to the beginning of an array
- shift removes and returns the first element

```perl
my @numbers = (1..9);
print @numbers . "n";

my @numbers = (1..100);
print @numbers*2 . "n";  # what happens here?
```

Scalar and list contexts

- What happens if you use a list in a place where a scalar is expected?
  - Perl will try to do the right thing, but might not do what you want.

```perl
my @numbers = (1..100);
print @numbers . "n";
```

```perl
# When used in a place that a scalar is expected,
#   Perl evaluates an array reference as the **number of elements in the array**

print ("number of elements of array = ", @numbers+0, "n");
print ("number of elements of array = ", scalar(@numbers), "n");
print ("number of elements of array = ", length(@numbers), "n");
```

Scalar and list contexts

- What happens if you use a list in a place where a scalar is expected?
  - Perl will try to do the right thing, but might not do what you want.
Scalar and list contexts

- What happens if you use a list in a place where a scalar is expected?
  - Perl will try to do the right thing, but might not do what you want.

```perl
my @numbers = (1..100);
!
# what happens here?
# each call, $i is assigned to one element.
foreach my $i (@numbers) {
  $i *= 2;
}
!
print “The sum is $sum”;
```

Looping

- Use `foreach` to loop over each element of an array
  - Syntax: `foreach my $i (@array) { code }
  - Code is called repeated, once for each element in array. In each call, $i is assigned to one element.

```perl
my @name = qw(dennis fil tony sally);
!
foreach my $person (@name) {
  print $person;
}
!```

Example: sum numbers from 1 to 100

- Using `foreach`
```perl
my $sum = 0;
foreach my $i (1..100) {
  $sum += $i;
}
!
print “The sum is $sum”;```

- Using `while`
```perl
my $sum = 0;
my $i = 1;
while ($i <= 100) {
  $sum += $i;
  $i++;
}
!
print “The sum is $sum”;```
Example: sum numbers from 1 to 100

```perl
// using until
my $sum = 0;
my $i = 1;
until ($i > 100) {
    $sum += $i;
    $i++;
}
print "The sum is $sum";
```

Example: sum numbers from 1 to 100

```perl
// using for
my $sum = 0;
for ($i=1; $i<=100; $i++) {
    $sum += $i;
}
print "The sum is $sum";
```

More complicated conditions

- Conjunction
  - condition && condition
- Disjunction
  - condition || condition
- Compare with bitwise operators
  - & computes the bitwise AND operation
  - | computes the bitwise OR operation

More complicated conditions

- Use parentheses when necessary
  ```perl
  ($a>1 && $b>1) || $c>1
  ```
  ```perl
  $a>1 && ($b>1 || $c>1)
  ```

Associative Arrays
a.k.a. Hashes

- Like an array, except that indices no longer have to be consecutive integers
- Can be strings or numeric values
- Hash variable names begin with %
- Refer to a specific element using $hash{$key}

```perl
$firstname{"Ryan"} = "Paul";
$firstname{"Biden"} = "Joe";
print $firstname{"Biden"};
```

Initializing hashes

```perl
# The following is a shortcut for these 2 lines:
# $name("Ryan") = "Paul";
# $name("Biden") = "Joe";
# syntax is %hash = (key1, value1, key2, value2, ...)
my %name = ("Ryan", "Paul", "Biden", "Joe");
print $name("Biden");
```
**Extracting lists of keys and values**

- Use `keys` and `values` to get a list of the keys and values of a hash.

```perl
my %course = {
    "I500" => "Fundamentals of Informatics",
    "I501" => "Introduction to Informatics",
    "I502" => "Information Management"
};

@k = keys %course;
foreach $i (@k) {
    print "$i
";
}

@v = values %course;
foreach $i (@v) {
    print "$i
";
}
```

**Number of elements in hash**

- How to count the number of `(key, value)` pairs in a hash?

```perl
my @k = keys(%course);
$count = $#k + 1;

// alternative...
$count = scalar(keys %course);
```

**Looping over Hashes**

```perl
foreach my $key (keys %course) {
    print "$key is 
";
}
```

- As an alternative, use the `each` operator.

```perl
while (my ($key, $value) = each %course) {
    print "$key is $value
";
}
```

**Checking Values**

- Use the `exists` function to check if a given key has an entry in the hash.

```perl
unless (exists $course{"I590") {
    $course{"I590"} = "Topics in Informatics";
}
```

**Deleting values**

- Delete the hash entry for a given key using `delete`, e.g.:

```perl
$id = "I590";
if (exists $course($id)) {
    delete $course($id);
}
```

**String comparison**

- Last time, we saw how to compare strings – Equals, greater than, less than, etc.

```perl
$A = <STDIN>;
chomp($A);
if ($A eq "yes") {
    print "The user agreed!
";
}
```

- What about more complicated comparisons? Enter Regular Expressions.
Regular Expressions

- Powerful “language” for describing strings
  - Using patterns
  - Patterns are written between slashes, /pattern/
  - Test whether a pattern matches a string using =~ operator
  - By default, this is true if any part of the string matches the pattern

- Example
  
  ```perl
  my $string = "Informatics";
  if ($string =~ /form/) {
    print "$string contains the substring "form"
  }
  
  Informatics contains the substring "form"
  ```

Metacharacters

- A period . matches any single character, except newline
  - For example, the pattern /fo.m/ matches:
    - informatics
    - foam
    - infoamatics
    - InFoRmAtics
  - But does not match:
    - InFoRmAtics
    - Infoamatics
    - fo
    - Foam
  - How to search for a period?

Metacharacters

- A star * matches any number of occurrences of the preceding symbol
  - Including zero matches

- /for*mr/ matches
  - form
  - foment
  - forrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrrr
Character classes

- A character class specifies a set of characters
  - Like \$ but restricts which characters can be matched
  - Specify using square brackets: \{ characters-to-match \}
  - As a shortcut, you can specify a range of characters using a hyphen: \{3-6\} is equivalent to \{3456\}

- Some examples:
  \{ a b c \}
  \{ a-z \}
  \{ a-f0-9 \}
  \{ . - \}

Negated character classes

- Alternatively, you can specify a character class by listing the characters you don’t want to match
  - Using the syntax \{ "characters-not-to-match" \}

- Some examples:
  \{ "abc" \}
  \{ "\" \}
  \{ "0-9" \}
  \{ "\\"\" \}

Anchors

- Recall that by default, a string s matches a pattern p
  if any substring of s matches p
  - E.g. “informatics” =~ /form/ is true
  - You can force exact matches by using anchors:
    \^ matches only the beginning of a string
    \$ matches only the end of a string
    \b matches only a word boundary

- Some examples:
  - Match any string beginning with Info: \^Info
  - Match any string ending with s: \s
  - Match a string consisting only of digits: \^[0-9]*\$