1.0.1.8.3

Introduction to Perl
Session 3

- lists and arrays
- for loop
- context

GOOD THING	IDIOM	BAD USAGE
do this for clarity and conciseness	Perlish construct	unless you are a donkey, don't do this

Recap

- scalar variables are prefixed by $ and can contain characters or numbers
- we saw the , as the list operator

```
print $a,$b,$c ;
($a,$b,$c) = (1,2,3) ;
```

a list

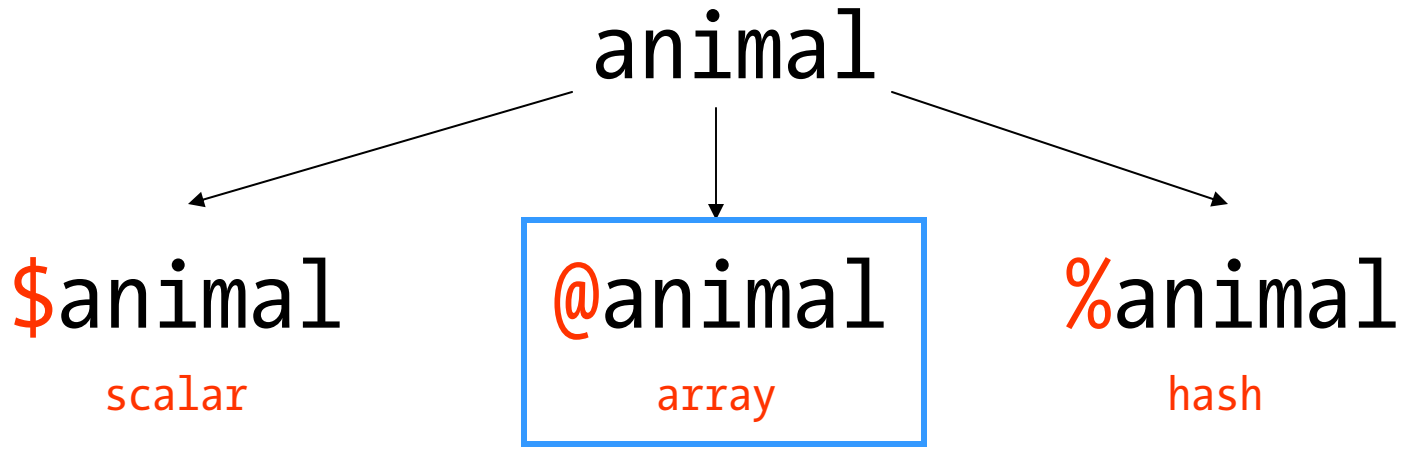
- recall substr(STR,OFFSET,LEN,NEWSTR) was used to isolate parts of a string, and
  - return a substring
  - replace the isolated substring with another string STR
  - if LEN=0 then NEWSTR is inserted
  - if LEN>0 and NEWSTR="" then part of STR is deleted

IDIOM

```
# deletes first 3 characters
substr($string,0,3,"");

# inserts $new at 5th character
substr($string,5,0,$new);
```

A New Variable – the Array

- recall that Perl variables are preceded by a character that identifies the plurality of the variable



- today we will explore the array variable, prefixed by @
- the variable "type" is array but the variable holds a list
  - remember the stretched soup in can analogy

Initializing Arrays

- to initialize the array, pass a list
  - we initialized a scalar by passing a single value

```
# $x is a scalar
$x = 2;

# @x is an array
@x = (1,2,3);
```

- an array variable is independent from a scalar variable of the same name
  - this is very important and can lead to confusion
  - arrays typically have plural names (@dogs vs @dog)


```
# while $dog and @dog are independent, different variables,
# their identical names can lead to confusion

$dog = “biff”;
@dog = (“biff”,”bark”,”howl”);
```

Quote Word Operator

- recall the use of qw( ) to easily define lists without typing quotes


```
# initialize three scalars
($x,$y,$z) = qw(biff bark howl);

# initialize an array
@dogs = qw(biff bark howl);
```

- qw() returns a list and it is natural to assign the output to an array

- what happens when you try to assign output of qw() to a scalar?

```
# assign a list to a scalar? we'll see the results shortly
$x = qw(biff bark howl);
```

Initializing with split

- remember split – the operator that broke up a string along a boundary


```
# split along any amoun of whitespace
$string = “a b  c   d     e”;
($a,$b,$c,$d,$e) = split(“ “,$string);
@letters = split(“ “,$string);

# split along a single character
$string = “a:b:c:d:e”;
@letters = split(“:”,$string);

# split along a string matching a regex
$string = “a1234b2332cd99310e”;
@letters = split(/\d+/,$string);
```

Initializing With a Range

- recall that we used a range of letters when defining a character class in regular expressions

```
# all letters a-to-z (a,b,c,...,z)
$is_match = $x =~ /[a-z]/;
```

- you can create a list made up of a range of numbers (successive values) using ..

```
(1..10)

equivalent to

(1,2,3,4,5,6,7,8,9,10)

but also

qw(1 2 3 4 5 6 7 8 9 10)
```

- num..num (1..10) or char..char (a..z)

Accessing Array Elements

- an array is an ordered set of elements
- elements are indexed by integers
- first element is indexed by 0 (0-indexing)
- if an array has n elements, last element is indexed by n-1

array variable	individual elements
@animals	$animals[0]
	$animals[1]
	$animals[2]
	…
	$animals[n-1]

Accessing Array Elements

- you may find the fact that the array is prefixed with @ but its elements are prefixed with $ counter-intuitive
  - you'll see why this is later – think *"arrays store lists of scalars"*

```
# an array of numbers 1 to 10
@nums = (1..10);

print $nums[0];   1
print $nums[1];   2
print $nums[2];   3
print $nums[9];   10

# $nums[10] is not defined, since @nums has 10 elements
print $nums[10];  ""

# settings element values
$nums[5] = 50;
$nums[6] = 60;

print $nums[5];               50
print $nums[6];               60
print $nums[5],$nums[6];      50 60
```

Negative Indexing

- recall that `substr` had facility to accept negative offsets to indicate distance from the end of the string
- array elements can be accessed similarly

```
# an array of numbers 1 to 10
@nums = (1..10);

# last element
print $nums[-1];  10

# second-last element
print $nums[-2];  9

# first and last elements
print $nums[0],$nums[-1];  1 10
```

Iterating Over an Array

- the for loop (foreach is a synonym) permits you to iterate across a list

```
@x = (1..5);

for $num (@x) {
  print $num," ",$num*$num,"\n";
}

1 1
2 4
3 9
4 16
5 25
```

- you will likely see foreach a lot, but I prefer the shorter for

```
foreach $num (@x) { CODE }    is the same as    for $num (@x) { CODE }
```

Iterating Over an Array

- you can iterate over the elements or array indices


```
@x = (1..5);

# iterate over elements
for $item (@x) {
  print $item,"\n";
}

# iterate over indices
for $i (0..4) {
  print $x[$i],"\n";
}
```

- choose the first approach if you don't need to determine an element's ordinal position

Iterating Over an Array

- a short script that prints the element of an array along with a *"this is the nth element"* string

IDIOM

```
@x = (1..5);

# iterate over indices
for $i (0..4) {
  print qq(This is the ${i}th element : $x[$i]);
}
```

this approach is preferred

```
@x = (1..5);

# iterate over elements, keep counter
$counter = 0;
for $num (@x) {
  print qq(This is the ${counter}th element : $num);
  $counter = $counter + 1:
}
```

this approach is unnecessarily verbose

Adding to an Array with Push

- there are many ways to add elements to an array
- the most common is push
  - push adds elements to the end of the array

```
@x = ();

# push single elements
push @x, 1;       # @x now (1)
push @x, 2;       # @x now (1,2)
push @x, 3;       # @x now (1,2,3)

# push a list of elements
push @x, 4, 5;    # @x now (1,2,3,4,5)
push @x, qw(6 7); # @x now (1,2,3,4,5,6,7)

@y = (8,9,10);
push @x, @y;      # @x now (1,2,3,4,5,6,7,8,9,10)
```

Initializing an Array with Push

- you can use for to initialize an array
- frequently used with push, which adds elements to the end of an array

```
@x = ();

for $num (1..10) {
  $num2 = $num*$num;
  push @x, $num2;
  print qq(added $num2, now last element is $x[-1]);
}

added 1, now last element is 1
added 4, now last element is 4
added 9, now last element is 9
...
added 100, now last element is 100
```

Arrays Grow as Necessary

- you may have noticed that we did not need to allocate memory for the array when we defined it
- the array variable grows and shrinks as necessary to accommodate new elements

```
@x = ();

$x[0] = 1;          # @x now (1)
$x[1] = 2;          # @x now (1,2)
$x[-1] = 3;         # @x now (1,3)

$x[3] = 4;          # @x now (1,2,undef,4)
```

- in this example we defined the 4th element, $x[3], *without explicitly definining the 3rd element*, $x[2] – Perl created memory space for $x[2] and set the value to undef

Arrays May Have undef Elements at End

- the last defined element marks the end of the array
  - this applies when initializing array elements with defined elements (i.e. not undef)

```
@x = ();

$x[5] = 5;          # @x now (undef,undef,undef,undef,undef,5)
$x[4] = 4;          # @x now (undef,undef,undef,undef,4,5)
```

- setting the last element to undef, does not shrink the array
  - memory is allocated, but contents are undefined

```
@x = (1..5);        # @x now (1,2,3,4,5)

$x[4] = undef;      # @x now (1,2,3,4,undef)
$x[3] = undef;      # @x now (1,2,3,undef,undef)
```

Shrinking an Array

- to extract the last element and shrink the array use pop

```
@x = (1..5);      # @x = (1,2,3,4,5);
$y = pop @x;      # $y = 5      @x = (1,2,3,4)
$y = pop @x;      # $y = 4      @x = (1,2,3)
$y = pop @x;      # $y = 3      @x = (1,2)
$y = pop @x;      # $y = 2      @x = (1)
$y = pop @x;      # $y = 1      @x = ()
$y = pop @x;      # $y = undef  @x = ()
```

- shift *is more popular than* pop, which extracts the first element, while also shrinking the array

```
@x = (1..5);      # @x = (1,2,3,4,5);
$y = shift @x;    # $y = 1      @x = (2,3,4,5)
$y = shift @x;    # $y = 2      @x = (3,4,5)
...
```

Arrays Grow and Shrink as Necessary

- in this example an array is created and then repeatedly elements are removed
  - one element removed with pop – from the back
  - one element removed with shift – from the front

```
@x = (1..10);

for $iteration (1..5) {
  my $x_popped = pop @x;
  my $x_shifted = shift @x;

  print qq(on iteration $iteration shifted $x_shifted and popped $x_popped);
}

on iteration 1 shifted 1 and popped 10
on iteration 2 shifted 2 and popped 9
on iteration 3 shifted 3 and popped 8
on iteration 4 shifted 4 and popped 7
on iteration 5 shifted 5 and popped 6
```

$#array

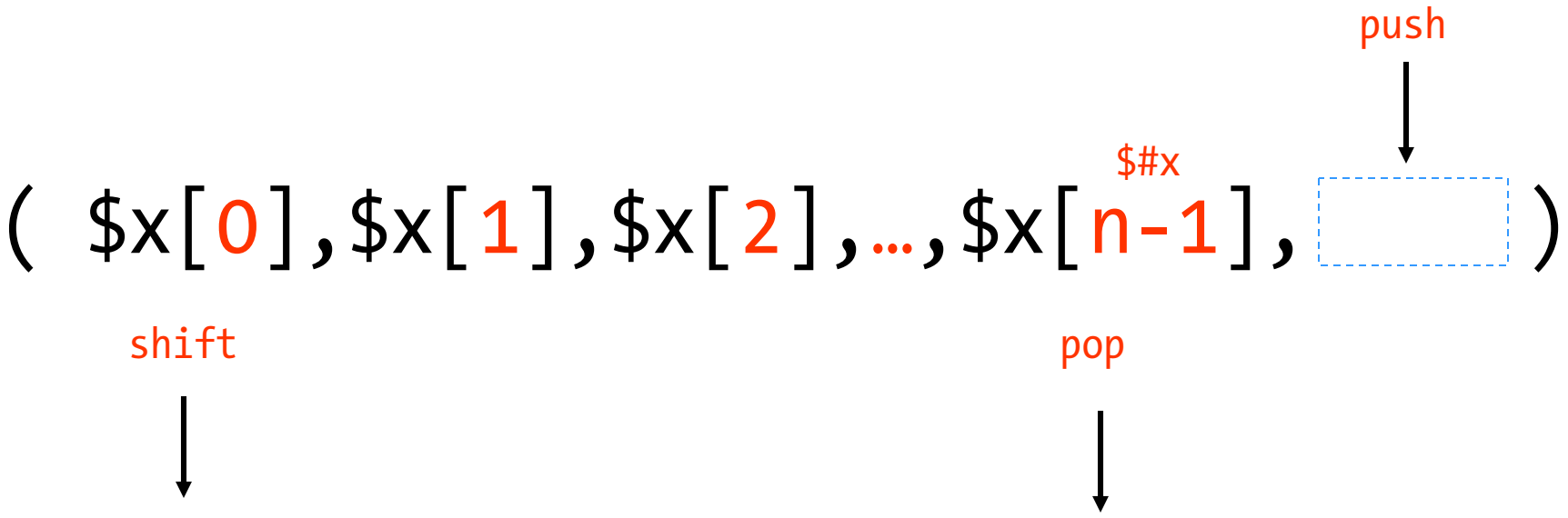
- what the $#@! is this?
- you've never seen this before, but you can guess what this variable holds
  - because it is prefixed by $, it holds a scalar value
  - $#array holds the index of the last element in the array


```
@x = (1..5);

$last_idx = $#x;              # $last_idx = 4

for $i (0..$last_idx) {
  print qq($i $x[$i]);
}

for $i (0..$#x) {
  print qq($i $x[$i]);
}
```

- I dislike $#array – it is too noisy
  - we'll see a cleaner alternative shortly

Manipulating Array Contents

- for now, these are the three ways to manipulate an array you need to be familiar with
  - remember that push can add a single element, or a list
  - shift/pop only remove one element at a time



Swapping Elements

- swapping elements is trivial – this may surprise you

```
# consider swapping the values of two scalars

$a = 5;
$b = 6;

($a,$b) = ($b,$a);

# apply the same to arrays

@x = (1,2);

( $x[0], $x[1] ) = ( $x[1], $x[0] );
```

- $x[1] is assigned to $x[0] and $x[0] is assigned to $x[1] **simultaneously**
  - there is no need for a temporary variable to hold one of the values
  - temp ← x0 ; x0 ← x1 ; x1 ← temp

Swapping Elements

```
@x = (1..5)                          # @x = (1,2,3,4,5)

($x[0],$x[-1]) = ($x[-1],$x[0]);     # @x = (5,2,3,4,1)

($x[0],$x[5]) = ($x[-1],6);          # @x = (1,2,3,4,1,6)
```

- let's randomly shuffle elements in an array by pair-wise swapping

```
@x = (1..10);

for $swap_count (1..5) {
  $i = int rand(10);  # random integer in range [0,9]
  $j = int rand(10);  # random integer in range [0,9]
  ( $x[$i], $x[$j] ) = ( $x[$j], $x[$i] );
  print qq(swapped $i $j array is now ) . join(" ",@x);
}

swapped 5 4 array is now 1 2 3 4 6 5 7 8 9 10
swapped 1 4 array is now 1 6 3 4 2 5 7 8 9 10
swapped 5 8 array is now 1 6 3 4 2 9 7 8 5 10
swapped 5 6 array is now 1 6 3 4 2 7 9 8 5 10
swapped 7 2 array is now 1 6 8 4 2 7 9 3 5 10
```

Introduction to Context

- make sure you are sitting comfortably – you are about to experience context
- context refers to the immediate code around a variable or operator that influences how the variable or operator are interpreted
- consider the following, in which we assign the output of a function to a scalar

```
$x = function();
```

- Perl has the facility to determine that we are assigning the result of function( ) to a scalar and can act accordingly
- the function could behave differently if we assign its output to an array

```
@x = function();
```

- for example, function($n) could return
  - in scalar context - number of perfect squares from 0..$n
  - in array context – the list of perfect squares from 0..$n

Introduction to Context

- what do you think happens in these two cases

```
# case 1                    # case 2
@y = @x                     $y = @x;
```

- in case 1, we are assigning an array to an array

  - Perl will copy the contents of array @x to array @y
  - the two arrays will have the same contents
  - the two arrays will be independent copies – changing one will not affect the other

- in case 2, we are assigning an array to a scalar

  - Perl interprets the array @x in scalar context
  - Perl returns the number of elements in @x
  - $y now holds the length of the array, @x

Determining the Length of an Array

- to obtain the number of elements in an array, evaluate it in scalar context


```
@x = (1..5);

# scalar ← array
$len = @x;

print “array has $len elements”;
```

- since arrays are o-indexed, an array with n elements has its last index n-1

```
@x = (1..5);

$len = @x;

for $i (0..$len-1) {
  print qq(The ${i}th element is $x[$i]);
}
```

$#x vs @x

- recall that $#x provided the index of the last element in an array
- @x in a scalar context gives the number of elements

```
$#x  is the same as   @x - 1
```

- @x-1 is easier on the eyes
- $#x has its uses, however
  - recall that substr( ) could extract parts of a string, but was also an *l-value*
  - well, $#x is also an *l-value*
  - you can assign a value to $#x to explicitly set the index of the last element, effectively growing/shrinking the array

IDIOM

```
@x = (1..5);
print $#x;          # 4
$#x = 5;            # @x = (1,2,3,4,5,undef)
$#x = 3;            # @x = (1,2,3,4)
$#x = 5;            # @x = (1,2,3,4,undef,undef)
```

More About Context

- context helps you write concise code – tread carefully

```
@x = (1..5);

# what is the value of $y?
$y = @x + 1;
```


```
@x = (1..5);

# why does this work?
for $i (0..@x-1) {
  print qq($i $x[$i]);
}
```


```
@x = (1..5);

# what is happening here? what is the last line printed?
for $i (0..@x) {
  print qq($i $x[$i]);
}
```

1.0.8.1.3

Introduction to Perl
Session 3

- you now know
  - all about arrays
  - declaring and initializing an array
  - growing and shrinking arrays
  - extracting elements and length of an array
  - for loop
  - iterating over arrays by element or index
  - application of split and join to arrays
  - context
- next time
  - hashes